PET-CoCE
computando culturas em equidade

Programação em Python e Introdução ao Pygame

Kaya Sumire Abe
kaya.sumire@gmail.com

UTFPR

Câmpus Curitiba – 2012
Versão 0.1

Versão rascunho, 0.1.
Colaboradores: Bianca Alessandra Visineski Alberton, Diego de Faria do Nascimento

Este trabalho foi confeccionado como material de suporte para as atividades do grupo PET Computando Culturas em Equidade.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

1 The Hard Way - o jeito mais difícil

Você deve pensar "o quê? Vamos aprender algo do jeito mais difícil? Não quero mais este curso!". Acalme-se. O jeito mais difícil, na verdade, é o mais fácil!

1.1 O que queremos dizer com "o jeito mais difícil"?

Não vamos abordar todos conceitos para depois colocar a mão na massa[1]. Isso torna qualquer coisa desinteressante! O que vamos fazer aqui é o seguinte:

1. Analisar cada exercício
2. Digitá-los cuidadosamente
3. Fazer cada um deles funcionar corretamente

Provavelmente, seguir estes passos com sucesso pode ser difícil, inicialmente. Começaremos com programas simples, para rever ou aprender conceitos básicos de programação. Durante as aulas, avançaremos nos conceitos e direcionaremos o que já foi apresentado para aplicações em jogos simples, mas que são o início para o desenvolvimento das aplicações mais sofisticadas do mercado.

As principais habilidades que necessitarão ser desenvolvidas neste curso são: leitura e escrita, atenção aos detalhes e localizar diferenças. Entretanto, ainda há outras recomendações a serem seguidas para uma boa prática da programação.

1.1.1 Leitura e Escrita

Pode parecer óbvio, mas se você possui problemas em digitação, você terá problemas em aprender a programar, principalmente se você tem problemas com caracteres especiais, comuns em programação, como chaves, colchetes, dois-pontos, etc.

Digitar os códigos apresentados e fazê-los funcionar, o ajudará a desenvolver esta habilidade.

1.1.2 Atenção aos detalhes

A habilidade que diferencia os bons programadores dos ruins é a atenção aos detalhes. Aliás, isto é o que diferencia o profissional bom do ruim em qualquer profissão. Sem prestar atenção nos mínimos detalhes do seu trabalho, você pode deixar passar elementos-chave das suas criações. Em programação, isto resulta em *bugs* e sistemas difíceis de utilizar.

Durante o curso, você copiará os códigos exatamente do jeito que eles forem apresentados. Isto fará com que você preste atenção nos detalhes.

1.1.3 Localizar diferenças

Uma habilidade muito importante, que programadores desenvolvem com a prática, é notar diferenças entre coisas, em especial, programas. Um programador experiente encontra diferenças entre códigos similares muito facilmente. Há ferramentas que fazem este "serviço" para os programadores, mas não utilizaremos nenhuma delas neste curso. Assim que treinarmos o cérebro, poderemos usar as ferramentas.

Enquanto você fizer os exercícios, digitando cada um, sem trapacear, haverá erros. É inevitável, mesmo programadores experientes os cometem. Seu trabalho é notar onde estão as diferenças entre o seu código e aquele que você estiver copiando. Fazendo isso, você treinará a si mesmo a identificar erros, *bugs* e outros problemas. Mais tarde, a tarefa será encontrar os erros nos seus próprios códigos, sem ter o que comparar.

[1] Mas vamos apresentar os conceitos fundamentais para construir qualquer programa!

1.1.4 Não copie e cole

Esqueça o *Ctrl + C* e o *Ctrl + V*. Você deve **DIGITAR** todos os exercícios. Do contrário, você estará enganando a si mesmo. O objetivo desta prática é treinar sua digitação de programas (que não é tão trivial quanto digitar um texto) e sua mente para ler, escrever e verificar um código.

1.1.5 Prática e persistência

Lembre-se que tudo que vale a pena, se é algo que você realmente quer, requer um certo esforço. Qualquer que seja o motivo por querer desistir, desista deste motivo. Force a si mesmo. Se você chegar em um exercício que não consegue resolver, esqueça-o por um momento e volte a resolvê-lo posteriormente. Traga suas dúvidas para a aula. Programação não é tão satisfatório no início, mas, depois, pode ser algo realmente prazeiroso.

Talvez, no início, você não entenda muita coisa, como se estivesse aprendendo qualquer idioma que você não conhece. Você se atrapalhará com as palavras, símbolos, etc., e, de repente, ocorrerá um "estalo" e tudo estará claro. Mas isto depende da persistência de cada um. Se você sempre fizer os exercícios e continuar tentando entendê-los, uma hora você os entenderá.

1.2 Você pode treinar em casa

Se você tem um computador em casa, nada lhe impede que você pratique os exercícios!

Aqui, não vamos ensiná-lo a instalar ou configurar o Python em seu computador. Há inúmeras páginas *web* que explicam cada etapa de instalação, em qualquer sistema operacional, e temos certeza de que você é suficientemente hábil para realizar esta tarefa.

2 Terminal interativo do Python

Tópicos abordados neste capítulo:

- Números inteiros e de ponto flutuante
- Expressões
- Valores e operadores
- Resolução de expressões
- Armazenamento de valores em variáveis

Antes de iniciarmos com os jogos, aprenderemos alguns conceitos básicos relacionados ao Python, tais como valores, operadores, expressões e variáveis. Não começaremos, de fato, a programar neste capítulo, mas aprenderemos a utilizar uma poderosa ferramenta: o terminal interativo do Python.

2.1 Matemática

Abra o terminal do Python de acordo com as orientações do instrutor. Em seu computador, você poderá fazer as operações aqui apresentadas através do terminal interativo de acordo com o sistema operacional utilizado.

Observe que há um cursor esperando para receber comandos. De início, digite $2 + 2$. Consequentemente, observe a resposta. Ou seja, também é possivel utilizar o terminal interativo do Python como uma calculadora.

Tabela 1: Operações matemáticas básicas.

2+2	adição
2-2	subtração
2*2	multiplicação
2/2	divisão

2.1.1 Números inteiros e de ponto flutuante

Em programação, assim como na matemática, há os números inteiros (tais como 6, 79, 22, -3, 0), também conhecidos como números "sem vírgula". Números fracionários ou "com vírgula", não são inteiros, como 3,2 e -7,89. No Python, 5 é um número inteiro, mas, se você defini-lo como 5.0[2], ele deixará de ser tratado como um inteiro, diferentemente da matemática convencional. Números com um ponto decimal são denominados **números de ponto flutuante**. Em programação, o computador considera qualquer número com ponto decimal como um não-inteiro.

Digite alguns destes problemas matemáticos no terminal interativo e pressione `Enter` depois de cada um deles. Observe os resultados.

$$2 + 2 + 2 + 2 + 2$$

$$8 * 6$$

$$10 - 5 + 6$$

$$2 \quad + \quad 2$$

Estes problemas são denominados *expressões*. Computadores podem resolver milhares de operações em tempo muito pequeno. Expressões são construídas com **valores** (os números) e conectadas por **operadores** (sinais matemáticos). Vamos analisar o que estes valores e operadores são.

[2] Observe que, em programação, a vírgula que utilizamos para definir números fracionários é representada por um ponto.

Figura 1: Uma expressão é construída com valores e operadores.

Como pode ser visto na última expressão da lista de exemplos anterior, é possivel colocar qualquer quantidade de espaços entre os inteiros e os operadores. (Entretanto, esteja certo de que não há espaços antes do início da expressão.)

Números são um tipo de valor enquanto os inteiros são tipos de números. Porém, embora os inteiros sejam números, nem todos os números são inteiros (por exemplo, 2.5)[3].

A seguir, veremos como trabalhar com texto em expressões. O Python não é limitado apenas a números, é muito mais que apenas uma calculadora!

2.2 Resolvendo expressões

Quando um computador resolve a expressão $10 + 5$ e resulta em 15, dizemos que ele resolveu a expressão. Ou seja, a expressão é reduzida a um valor único, igual a resolução de um problema em matemática, reduzindo-o para um único número: a resposta.

Expressões como $10 + 5$ e $10 + 3 + 2$ possuem o mesmo valor, pois ambas resultam no mesmo resultado. Mesmo valores únicos são considerados expressões: a expressão 15 resulta no valor 15.

Entretanto, se você digitar apenas 5+, no terminal interativo, você obterá uma mensagem de erro:

```
>>> 5 +
SyntaxError: invalid syntax
```

Este erro ocorre porque 5+ não é uma expressão. Expressões contém valores conectados por operadores, e + é um operador que espera dois itens a serem conectados no Python. Como apenas definimos um valor, a mensagem de erro apareceu. Um erro de sintaxe (`syntax error`) significa que o computador não entendeu a instrução dada, por ter sido digitada incorretamente. O Python sempre mostrará uma mensagem de erro caso ele não consiga entender alguma expressão.

Isto pode não parecer tão importante, inicialmente. Contudo, programação não é apenas `dizer ao computador o que fazer`: é dizer **corretamente** o que um computador deve fazer. Do contrário, como esperado, ele não executará as ordens corretamente.

2.2.1 Expressões dentro de expressões

Expressões também podem conter expressões. Por exemplo, na expressão $2 + 5 + 8$, "$2 + 5$" também é uma expressão. $2 + 5$ é resolvido pelo Python, gerando o valor 7 e a expressão original, então, se torna $7 + 8$, finalmente gerando o resultado 15.

2.3 Armazenando valores em variáveis

Quando programamos, frequentemente queremos armazenar valores, gerados pelas expressões ou não, para utilizá-los posteriormente no programa. Podemos armazenar valores em **variáveis**.

[3] É a mesma analogia de que gatos são animais, mas nem todos os animais são gatos.

Pense em variáveis como caixas que podem guardar valores. Você pode armazenar valores em variáveis utilizando o sinal =, denominado **sinal de atribuição**. Por exemplo, para armazenar o valor 15 na variável `spam`, entre com `spam = 15` no terminal.

```
>>> spam = 15
```

Continuando com a analogia da caixa, conforme ilustrado na figura 2, `spam` é a etiqueta (ou como a caixa é nomeada) e o valor é um pequeno item guardado na caixa.

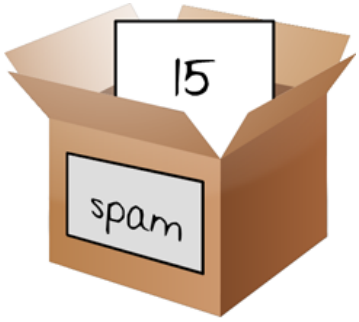

Figura 2: Variáveis são como caixas que podem guardar valores.

Quando `Enter` for pressionado, não haverá nada a ser respondido, ao contrário do que acontecia com os resultados das operações anteriores. Se não houver mensagens de erro, aparecerá apenas uma linha em branco. Isto quer dizer que a **instrução** foi executada com sucesso. O próximo >>> aparecerá para que a próxima instrução seja digitada.

Esta instrução, denominada **sentença de atribuição**, cria a variável `spam` e armazena o valor 15 nela. Diferente das expressões, sentenças são instruções que não chegam a valores finais (como resultados dos problemas matemáticos), motivo pelo qual o terminal não mostra valores na linha seguinte a elas.

Pode parecer confuso entender qual é a diferença entre sentença e expressão. O segredo é: se uma instrução gera um valor final, então é uma expressão. Senão, então é uma sentença.

Uma sentença de atribuição é gerada por um nome para a variável, seguido de um sinal = seguido por uma expressão. O valor que a expressão gera é armazenado na variável. Logo, as expressões vistas anteriormente podem ser armazenadas em variáveis.

Lembre-se que variáveis armazenam valores, não expressões. For exemplo, se houver a instrução `spam = 10 + 5`, a expressão 10+5 será resolvida e o resultado 15 será armazenado na variável `spam`.

Para verificar se a variável foi criada corretamente, digite o `spam` no terminal. O resultado esperado é o seguinte:

```
>>> spam = 15
>>> spam
15
```

Agora, `spam` retorna o valor dentro da variável, 15. Aqui há uma interessante mistura: observe o resultado de `spam + 5`.

```
>>> spam = 15
>>> spam
15
>>> spam + 5
20
>>>
```

Isto pode parecer estranho, mas faz sentido ao lembrarmos que o valor de `spam` é 15. Devido à `spam` ter este valor armazenado, executar `spam + 5` naturalmente resulta em 20, como a expressão $15 + 5$.

Se você não tiver criado uma variável antes de utilizá-la, o Python retornará uma mensagem de erro, pois a variável em questão não existe. Isto também acontece se você digitar o nome de uma variável incorretamente.

É possível mudar o valor armazenado em uma variável entrando com uma nova sentença de atribuição. Por exemplo, tente o seguinte:

```
>>> spam = 15
>>> spam + 5
20
>>> spam = 3
>>> spam + 5
8
>>>
```

A primeira expressão resulta em 20, pois armazenamos o valor 15 em `spam`. Então, entramos com `spam = 3`, e o valor 15 é substituído pelo valor 3. Agora, se entrarmos novamente com `spam + 5`, o valor será outro, no caso, 8, pois `spam` agora vale 3.

Para descobrir qual o valor de uma variável, apenas digite o nome da variável no terminal, como feito anteriormente.

Agora, algo interessante: devido à uma variável ser apenas um **nome** para um valor, podemos escrever expressões como:

```
>>> spam = 15
>>> spam + spam
30
>>> spam - spam
0
>>>
```

Estas expressões usam a variável `spam` duas vezes cada. Você pode usar variáveis quantas vezes quiser, ou for necessário. Lembre-se que o Python calculará as expressões com os valores armazenados nelas, cada vez que uma variável for utilizada.

Desta forma, podemos usar o valor de uma variável para atribuir um novo valor a ela mesma. Observe.

```
>>> spam = 15
>>> spam = spam + 5
20
>>>
```

A sentença de atribuição `spam = spam + 5` é o mesmo que "o novo valor da variável `spam` é o valor atual mais 5". Lembre-se que a variável do lado esquerdo do sinal = armazenará o valor que a expressão, do lado direito do sinal, vale. Assim, podemos continuar aumentando o valor de `spam` do jeito e quantas vezes quisermos.

```
>>> spam = 15
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam
30
>>>
```

2.3.1 Sobreencrevendo valores das variáveis

Mudar o valor de uma variável é fácil: basta formular outra sentença de atribuição com a mesma variável. Veja o que acontece com o seguinte código dado como entrada no terminal:

```
>>> spam = 42
>>> print(spam)
42
>>> spam = 'Hello'
```

```
>>> print(spam)
Hello
>>>
```

Inicialmente, `spam` recebe o valor 42. Esta é a forma com que o primeiro comando `print` imprime 42. Mas, quando `spam = 'Hello'` é executado, o valor 42 é "esquecido" pela variável e o `'Hello'` fica no lugar do valor anterior na variável `spam`.

Substituir um valor em uma variável é frequentemente denominado **sobreescrevê-lo**. É importante salientar que o valor antigo da variável é perdido permanentemente. Se for necessário utilizar tal valor futuramente, defina uma nova variável para recebê-lo.

```
>>> spam = 42
>>> print(spam)
42
>>> oldSpam = spam
>>> spam = 'Hello'
>>> print(spam)
Hello
>>> print(oldSpam)
42
```

No exemplo anterior, antes de sobrescrevermos o valor de `spam`, armazenamos o valor atual em uma variável chamada `oldSpam`.

2.4 Usando mais de uma variável

Quando os programas tomarem forma mais complexa do que estes do terminal interativo do Python, não é conveniente que haja limitação para o uso de apenas uma variável, certo? Logo, veremos como lidar com mais de uma variável.

Por exemplo, vamos declarar duas variáveis:

```
>>> fizz = 10
>>> eggs = 15
```

Agora, `fizz` possui o valor 10 e `eggs`, o valor 15.

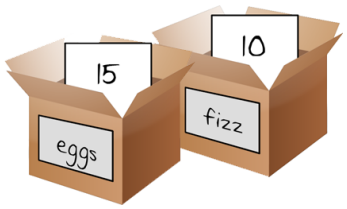

Figura 3: As variáveis `fizz` e `eggs` ilustradas.

Sem mudar o valor da variável `spam`, vamos atribuir um novo valor a ela. Entre com `spam = fizz + eggs` no terminal e depois verifique o valor de `spam`. Você consegue imaginar qual é o resultado?

```
>>> fizz = 10
>>> eggs = 15
>>> spam = fizz + eggs
>>> spam
25
>>>
```

O valor de `spam` agora é 25, porque somamos `fizz` e `eggs`, logo, somamos os valores armazenados em `fizz` e em `eggs`, e armazenamos o resultado em `spam`.

2.5 Resumo

Neste capítulo, foi apresentado o básico sobre instruções do Python. É necessário que seja dito exatamente o que deve ser feito, porque computadores não possuem "senso comum" e apenas "entendem" instruções muito simples. Você aprendeu que o Python pode resolver expressões (isto é, reduzir uma expressão a um valor único) e que expressões são valores combinados com operadores (como + ou −). Você também aprendeu que é possível armazenar valores em variáveis para utilizá-las posteriormente.

No próximo capítulo, veremos mais conceitos fundamentais e estaremos prontos para programar!

2.6 Exercícios complementares

1. Usando o terminal interativo faça:
   (a) Atribua quatro notas a quatro variáveis diferentes (`nota1`, `nota2`, `nota3` e `nota4`), atribua a média desses valores à variável `media`, e imprima o resultado na tela.
   (b) Atribua o valor de π (3,1415...) à variavel `pi` e um valor positivo qualquer à variável `raio`, após calcule o comprimento da circunferência ($2\pi R$) e a área do circulo (πR^2)

3 Strings

Tópicos abordados neste capítulo:

- Ordem de execução
- Strings
- Concatenação de strings
- Tipos de dados
- Função *print*
- Função *input*
- Comentários
- Nomear variáveis adequadamente
- Case-sensitivity
- Sobreescrevendo variáveis
- Comandos para caracteres especiais

Já vimos o suficiente sobre números no capítulo anterior. Python é mais do que uma simples calculadora. Agora, vamos ver o que é possível fazer com textos. Neste capítulo, aprenderemos a armazenar texto em variáveis, combinar textos e mostrá-los na tela. Muitos dos programas que veremos usarão textos para serem mostrados na tela para o jogador/usuário e o mesmo deverá entrar com algum texto através do teclado. Também construiremos o primeiro código fora do terminal interativo.

3.1 Strings

No Python(e talvez em todas as linguagens de programação), denominamos pequenos trechos de texto como **strings**. Podemos armazenar valores *string* da mesma forma que armazenamos números. Quando digitamos strings, elas deverão estar entre aspas simples (’), como o seguinte:

```
>>> spam = ‘hello’
>>>
```

As aspas simples servem para informar o computador onde a string começa e termina e não faz parte do valor da string. Se você digitar `spam` no terminal, você verá o conteúdo da variável `spam` (no caso anterior, a string ’hello’).

Strings podem ter (quase) quaisquer caracteres ou sinais, assim como espaços e números:

```
‘hello’
‘Hi there!’
‘Albert’
‘KITTENS’
‘7 apples, 14 oranges, 3 lemons’
‘A long time ago in a galaxy far, far away...’
‘O*&#wY\%*&OCfsdYO*&gfC\%YO*&\%3yc8r2’
```

3.1.1 Concatenação de strings

Você pode ligar uma string a outra utilizando o operador +, o que é chamado de concatenação de strings. Observe:

```
>>> ‘Hello’ + ‘World!’
‘HelloWorld!’
>>>
```

Uma alternativa para manter as strings separadas é adicionar um espaço depois de ‘Hello’:

```
>>> 'Hello ' + 'World!'
'Hello World!'
>>>
```

Strings e inteiros são diferentes **tipos de dados**. Todos os valores possuem um tipo definido. O tipo de valor de `'Hello'` é *string*, enquanto o tipo de 5 é inteiro.

3.2 Escrevendo o primeiro script

Em Python, geralmente chamamos os programas fora do terminal interativo de **scripts**, apenas por conveniência. Você pode utilizar qualquer editor de texto que não adote formatação ao salvar um arquivo para escrever um script. E você pode escrever o script no terminal também, entretanto não é confortável sempre escrever o mesmo programa linha a linha, se ele for muito longo. Por exemplo, você pode usar o bloco de notas do Windows ou o gedit, no Ubuntu. Basta salvar o arquivo com a extensão .py. Se você está em dúvida ou não sabe por onde começar, esteja atento às instruções em sala de aula ou pesquise sobre isso em casa.

3.2.1 "Hello World!"

Tradicionalmente, quando aprendemos uma nova linguagem, o primeiro programa feito é mostrar a saudação "Hello World!"[4] na tela. Criaremos o nosso próprio programa "Hello World!" agora.

Quando você entrar com o programa, não digite os números do lado esquerdo do código. Eles servem apenas para que possamos nos localizar mais facilmente em um código e não fazem parte dele como um todo. Digite o texto a seguir no editor de sua preferência. Chamamos este "texto" de *código fonte do programa*, pois ele contém as instruções que o Python seguirá para determinar o comportamento do programa.

Certifique-se que estará usando a versão Python 3, ou o programa não funcionará.

Código 1: Primeiro script.

```
1 # Este programa me cumprimenta e pede o meu nome.
2 print('Hello world!')
3 print('What is your name?')
4 myName = input()
5 print('It is good to meet you, ' + myName)
```

Se o programa não funcionar, leia o erro e tente consertá-lo, se for um erro de sintaxe. Caso obtenha uma mensagem similar a:

```
Hello world!
What is your name?
Albert

Traceback (most recent call last):
File "C:/Python26/test1.py", line 4, in <module>
myName = input()
File "<string>", line 1, in <module>
NameError: name 'Albert' is not defined
```

Quer dizer que a versão que você está utilizando, do Python, não é a correta para a execução deste programa. Verifique a compatibilidade do Python 2 em relação ao Python 3 e conserte o que for necessário, ou certifique-se que está usando a versão 3.

Tarefa 3.1
Pesquise sobre as diferenças entre as versões 2 e 3 do Python e faça as alterações necessárias para o código 1 funcionar no Python 2.

[4] "Olá, mundo!"

3.3 Como o programa "Hello World" funciona

Como este programa funciona? Cada linha em que demos entrada é uma instrução ao computador que é interpretada pelo Python, de forma que o computador entenda. Um programa de computador é como uma receita de bolo: primeiro passo, segundo passo, e assim por diante, até chegar ao fim. Cada instrução é seguida em sequência, do início ao fim do programa.

O passo-a-passo de um programa é chamado de **ordem de execução** ou apenas **execução**, para simplificar.

Agora vamos analisar minuciosamente este programa.

3.3.1 Comentários

```
1 # Este programa me cumprimenta e pede o meu nome.
```

Esta linha é um **comentário**. Qualquer texto seguido por um sinal de sustenido (#) é um comentário. Comentários não são para o computador, mas para o programador; o computador os ignora. Eles são utilizados para lembrar ao programador ou informá-lo sobre o que o código faz.

3.3.2 Funções

Uma **função** é como se fosse um mini-programa dentro do seu programa, contém linhas de código que são executadas do início ao fim. O Python fornece algumas funções prontas para facilitar a prática da programação. A vantagem disso é que não precisamos saber dos passos que a função toma para executar determinada ação, apenas como utilizá-la para obter um resultado.

Uma **chamada de função** é um trecho de código que informa ao programa para executar o código dentro da função. Por exemplo, o seu programa pode chamar a função `print()` sempre que você quiser mostrar algum texto na tela.

A função `print()`

```
2 print('Hello world!')
3 print('What is your name?')
```

Estas linhas chamam a função `print()`, com a string a ser impressa dentro dos parênteses.

Adicionamos parênteses no fim dos nomes das funções para deixar claro que estamos nos referindo a funções e não a variáveis que possam ter o mesmo nome de uma função. Analogamente, utilizamos aspas para nos referirmos às strings: por exemplo, ao utilizarmos '42' nos referimos à string 42 e não ao inteiro 42.

Comandos para caracteres especiais

Eventualmente, gostaríamos de imprimir alguns caracteres que são vistos na programação como caracteres especiais, tais como barras, aspas, apóstrofos, etc. Como eles não são caracteres que possam ser impressos normalmente com a função `print()`, então utilizamos alguns comandos simples para utilizá-los em modo texto[5]. Outros recursos também podem ser utilizados na função `print()` para pular linhas ou imprimir tabulações. Observe a tabela 2.

Aspas duplas e aspas simples

Strings não necessitam vir sempre entre aspas simples, no Python. É possível colocar texto entre aspas duplas também, obtendo-se o mesmo efeito.

```
>>> print('Hello world')
Hello world
>>> print("Hello world")
Hello world
```

[5] Em inglês, chamados de **escape characters**.

Tabela 2: Comandos para caracteres especiais.

Comando	O que é impresso
\\	Contra-barra (\)
\'	Aspas simples (')
\"	Aspas duplas (")
\n	Nova linha
\t	Tabulação

Entretanto, não se deve misturar aspas simples e duplas.

```
>>> print('Hello world")
SyntaxError: EOL while scanning
single-quoted string
```

Lembre-se que só é necessário utilizar o comando \' quando uma string é envolvida por aspas simples. O mesmo serve para as aspas duplas. Observe o exemplo.

```
>>> print('I asked to borrow Abe\'s car for a week. He said, "Sure."')
I asked to borrow Abe's car for a week. He said, "Sure."
>>> print("He said, \"I can't believe you let him borrow your car.\"")
He said, "I can't believe you let him borrow your car."
```

A função `input()`

```
4 myName = input()
```

Esta linha possui uma sentença de atribuição à uma variável (`myName`) e uma chamada de função (`input()`). Quando a função `input()` é chamada, o programa espera por uma entrada (input) do usuário com texto. A string que o usuário entra (neste caso, seu nome) torna-se o valor de saída (`output`).

Como expressões, chamadas de função resultam em um valor único. Este resultado é chamado de **valor de retorno**. Neste caso, o valor de retorno da função `input()` é a string dada como entrada.

Apesar do seu nome, a função `input()` não necessita de nenhuma entrada, ou, como chamamos em programação, **argumento**, que são dados dentro dos parênteses.

Na última linha do programa, temos, novamente, uma função `print()`. Desta vez, utilizamos o operador + para concatenar a string `'It is good to meet you '` e a string armazenada na variável `myName`, que contém o nome que o usuário deu como entrada ao programa. Esta é a forma com que o programa nos cumprimenta pelo nome.

3.3.3 Finalizando o programa

Uma vez que o programa executa a última linha, ele para. Neste ponto, ele **termina** o programa e todas as variáveis são esquecidas pelo computador, inclusive a string que armazenamos na variável `myName`. Se você tentar executar o programa novamente, com um nome diferente, o novo nome será utilizado.

```
Hello world!
What is your name?
Carolyn
It is good to meet you, Carolyn
```

Lembre-se que o computador executa exatamente o que for programado para fazer. Computadores são burros. Este programa não se importa com o que você digitar no lugar do seu nome. Você pode digitar qualquer coisa e o computador o tratará da mesma forma:

```
Hello world!
What is your name?
poop
It is good to meet you, poop
```

3.4 Nome de variáveis

O computador não se importa com o nome que você dá para as suas variáveis. Entretanto, o recomendado é que as variáveis possuam nomes que facilitem o seu entendimento.

A identificação dos nomes de variáveis (assim como qualquer outra coisa no Python) diferenciam maiúsculas de minúsculas (**case-sensitive**). Isso significa que uma variável com o mesmo nome, mas com a escrita diferente, entre maiúsculas e minúsculas, não são consideradas a mesma coisa. Assim, `spam`, `SPAM` e `sPAm` são consideradas diferentes variáveis, cada uma podendo carregar um valor diferente.

Contudo, não é uma boa ideia utilizar variáveis com o mesmo nome, para evitar confusões. Por exemplo, uma simples troca de variável pode fazer com que o resultado não seja o esperado, embora o programa execute corretamente. Este erro é frequentemente chamado de **bug** e é comum ocorrerem acidentes deste tipo enquanto construímos programas. Por isso, é importante que os nomes das variáveis que você escolher façam sentido.

É útil usar letras maiúsculas em variáveis quando é utilizado mais de uma palavra em seu nome. Por exemplo, se você utilizar `nomeDaVariavel` ao invés de `nomedavariavel` pode tornar o código mais fácil de ser lido. Esta é uma **convenção** (ou seja, uma padronização opcional para se trabalhar) da programação em Python e em outras linguagens também.

3.5 Resumo

Agora que você aprendeu a lidar com texto, podemos começar a construir programas com os quais podemos interagir. Isto foi importante porque é através do texto que o usuário e o computador se comunicarão.

Strings são um diferente tipo de dado que utilizaremos em nossos programas. Podemos utilizar o operador + para concatenar strings.

No próximo capítulo, vamos aprender mais sobre como lidar com variáveis para que o programa utilize os dados e números dados como entrada em um programa. Uma vez que tenhamos aprendido como utilizar texto, números e variáveis, estaremos prontos para criar jogos.

3.6 Exercícios complementares

1. Faça um programa (script) que peça dois números ao usuário e imprima a soma.
2. Faça um programa que peça um lado do quadrado, calcule a área, em seguida mostre o dobro desta área para o usuário.
3. Faça um programa que peça a temperatura em graus Fahrenheit, transforme e mostre a temperatura em graus Celsius. $(C = (5 * (F - 32)/9))$.
4. Faça um programa que peça 2 números inteiros e um número real. Calcule e mostre:
   (a) O produto do dobro do primeiro com metade do segundo .
   (b) A soma do triplo do primeiro com o terceiro.
   (c) O terceiro elevado ao cubo.
5. Faça um programa que peça o tamanho de um arquivo para download (em MB) e a velocidade de um link de Internet (em Mbps), calcule e informe o tempo aproximado de download do arquivo usando este link (em minutos). Obs. 1 MB (megabyte) = 8 Mb(megabit).

4 Adivinhe o número

Tópicos abordados neste capítulo:

- Sentenças *import*
- Módulos
- Argumentos
- Sentenças *while*
- Condições
- Blocos
- Valores booleanos
- Operadores de comparação
- Diferença entre = e ==
- Sentenças *if*
- A palavra-chave *break*
- As funções *str()* e *int()*
- A função *random.randint()*

4.1 O jogo "adivinhe o número"

Através do que já vimos, vamos fazer um jogo de adivinhação de números. Neste jogo, o computador sorteará um número entre 1 e 20 e pedirá ao usuário para adivinhar este número. Você terá 6 chances e o computador o informará se seu palpite é maior ou menor que o número sorteado. Se você adivinhá-lo dentro de 6 tentativas, você vence.

Este é um bom programa de início, pois utiliza números (pseudo) aleatórios, *loops* e entrada do usuário em um código curto. Assim que você escrever o programa, você aprenderá a converter valores para diferentes tipos de dados (e o porquê de necessitar este artifício).

Devido a estarmos lidando com jogos, eventualmente chamaremos o usuário de **jogador**, mas chamá-lo de usuário também é adequado.

4.2 Modelo de execução do "adivinhe o número"

É desta forma que o jogo aparecerá ao jogador assim que o programa rodar.

```
Hello! What is your name?
Albert
Well, Albert, I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too high.
Take a guess.
2
Your guess is too low.
Take a guess.
4
Good job, Albert! You guessed my number in 3 guesses!
```

4.3 Código-fonte

Nesta seção você encontra o código fonte para este jogo. Se você já sabe um pouco de programação, através do que já foi abordado, você pode tentar implementar este jogo antes de ver o código. Entretanto, se você não tem ideia de como começar, digite cuidadosamente o código 2.

Código 2: Jogo "adivinhe o número".

```
1  # Jogo ``adivinhe o numero''.
2  import random
3
4  guessesTaken = 0
5
6  print('Hello! What is your name?')
7  myName = input()
8
9  number = random.randint(1, 20)
10 print('Well, ' + myName + ', I am thinking of a number between 1 and 20.')
11
12 while guessesTaken < 6:
13     print('Take a guess.')
14     guess = input()
15     guess = int(guess)
16
17     guessesTaken = guessesTaken + 1
18
19     if guess < number:
20         print('Your guess is too low.')
21
22     if guess > number:
23         print('Your guess is too high.')
24
25     if guess == number:
26         break
27
28 if guess == number:
29     guessesTaken = str(guessesTaken)
30     print('Good job, ' + myName + '! You guessed my number in ' +
31         guessesTaken + ' guesses!')
32
33 if guess != number:
34     number = str(number)
35     print('Nope. The number I was thinking of was ' + number)
```

Importante! Tenha certeza de que você executará o programa com o Python 3. Se você utilizar o Python 2, provavelmente encontrará erros na maioria dos programas. É possível fazer algumas alterações para rodar os programas, mas é mais fácil que você esteja usando a versão correta!

Se o programa não funcionar depois de você tê-lo digitado, verifique se tudo foi digitado corretamente.

4.4 A sentença `import`

Depois da primeira linha de comentário, que nos informa o que o programa faz, temos uma linha de importação:

```
2 import random
```

Esta é uma **sentença de importação**. Sentenças não são funções (observe que nem `import` ou `random` possuem parênteses em seguida). Lembre-se que sentenças são instruções que realizam alguma ação mas não geram resultado. Já discutimos sentenças: sentenças de atribuição armazenam um valor em uma variável (mas a sentença em si, não gera resultados, como operações fazem).

Enquanto o Python possui muitas funções, algumas delas existem em programas separados, chamados **módulos**. Módulos são programas do Python que contém funções adicionais. Utilizamos funções destes módulos trazendo-os para os programas com uma sentença `import`. Neste caso, estamos importando o módulo `random`.

A sentença `import` é construída através do comando `import` seguido pelo nome do módulo desejado. A linha 2 do programa para adivinhar um número possui uma sentença destas, que traz o

módulo `random` que contém várias funções relacionadas à números aleatórios (randômicos). Utilizamos uma de suas funções para gerar o número a ser adivinhado.

```
4 guessesTaken = 0
```

Esta linha cria uma nova variável chamada `guessTaken`. Vamos armazenar o número de tentativas, que o jogador fará, nesta variável. Já que, no início do programa, não temos nenhuma tentativa, então iniciamos esta variável com o valor 0.

```
6 print('Hello! What is your name?')
7 myName = input()
```

As linhas 6 e 7 são as mesmas que utilizamos no programa anterior, o "Hello World", visto anteriormente. Programadores frequentemente reutilizam trechos de código de outros programas quando eles precisam de fazer algo que eles já fizeram anteriormente.

A linha 6 é uma chamada para a função `print()`. Lembre-se que uma função é como um mini-programa rodando, e, então, quando o programa chama uma função, ele roda este mini-programa. O código dentro da função `print()` mostra a string que lhe foi passada dentro dos parênteses na tela.

Quando estas linhas terminam de executar, a string que for designada como o nome do jogador será armazenada na variável `myName`.

4.5 A função `random.randint()`

```
9 number = random.randint(1, 20)
```

Na linha 9, chamamos uma nova função, `randint()`, e então armazenamos o seu valor em uma variável chamada `number`. Lembre-se que chamadas de funções são expressões porque elas geram resultado. Chamamos este resultado de **retorno da função**.

Devido à função `randint()` ser fornecida pelo módulo `random`, ela é precedida pelo nome do módulo seguindo de um ponto, para sinalizar que a função pertence a tal módulo. A função `randint()` retorna um número aleatório, inteiro, entre e inclusive os números que estipulamos dentro dos parênteses. Neste caso, fornecemos os números 1 e 20, separados por uma vírgula. O número gerado pela função é armazenado na variável `number`.

Por um momento, vamos voltar ao terminal interativo e entrar com `import random` para importar o módulo `random`. Entre com `random.randint(1, 20)` para observar o resultado da chamada da função. Ela deve retornar um inteiro entre 1 e 20, inclusive. Agora, entre com a mesma linha, novamente. Ele poderá resultar em um número inteiro diferente. Isto ocorre porque a cada vez que a função `randint()` é chamada, ela retorna um número aleatório, como se estivéssemos lançando um dado.

```
>>> import random
>>> random.randint(1, 20)
12
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
3
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
7
>>>
```

Sempre que quisermos adicionar alguma aleatoriedade ao código, esta função pode ser muito útil para casos gerais. Você também pode mudar a faixa de valores, dependendo da sua necessidade.

Por exemplo, se eu quiser uma faixa de números entre 1 e 100, posso modificar as linhas 9 e 10 para as seguintes:

```
9  number = random.randint(1, 100)
10 print('Well, ' + name + ', I am thinking of a number between 1 and 100.')
```

E, agora, o programa sorteará um número entre 1 e 100, inclusive.

4.5.1 Chamando funções que pertencem a módulos

Esteja certo de que, ao chamar a função para gerar um número aleatório, você tenha chamado-a `random.randint(1, 20)` e não apenas `randint(1, 20)` ou o computador não saberá que a função pertence ao módulo `random` e poderá ocorrer um erros similar a este:

```
>>> randint(1, 20)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'randint' is not defined
>>>
```

Lembre-se que seu programa precisa importar um módulo antes de utilizar as funções dele (senão, como o computador vai saber onde procurar o que você precisa?). Por isso que as sentenças de importação de módulos geralmente são declaradas no início do programa.

4.6 Passando argumentos para funções

Os valores inteiros entre os parênteses da chamada de função `randon.randint(1, 20)` são chamados de **argumentos**. Argumentos são valores passados a uma função, quando esta é chamada. Eles "dizem" è função como se comportar. Assim como as entradas (inputs) do usuário alteram o comportamento do programa, argumentos são entradas para funções.

Algumas funções exigem que valores sejam passados à elas. Por exemplo:

```
input()
print('Hello')
random.randint(1, 20)
```

A função `input()` não possui argumentos, mas a função `print()` possui um e a função `randint()` possui dois. Quando mais de um argumento é exigido, estes devem ser separados por vírgulas, também conhecidas como **delimitadores**. É através deste recurso que o computador sabe onde termina um valor e inicia outro.

Se você passar mais (ou menos) argumentos necessários para a função, o Python disparará uma mensagem de erro. Por exemplo, se apenas um valor for fornecido à função `randint()`:

```
>>> random.randint(1)
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
random.randint(1)
TypeError: randint() takes exactly 3 positional
arguments (2 given)
>>> random.randint(1, 2, 3)
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
random.randint(1, 2, 3)
TypeError: randint() takes exactly 3 positional
arguments (4 given)
>>>
```

4.7 Saudando o jogador

As linhas 10 e 12 cumprimentam o jogador e o informam sobre o jogo, permitindo, então, que o jogador adivinhe o número secreto. A linha 10 é bem simples, mas a linha 12 inicia um conceito muito importante, chamado laço de repetição.

```
10 print('Well, ' + myName + ', I am thinking of a number between 1 and 20.')
11
12 while guessesTaken < 6:
```

4.8 Laços de repetição (*loops*)

A linha 12 possui um laço `while`, que indica o início de *loop*. **Loops** são partes do código que são executadas inúmeras vezes. Entretanto, antes de aprendermos sobre *loops* no Python[6], vamos ver alguns conceitos.

4.9 Blocos

Um **bloco** é definido como uma ou mais linhas de códigos agrupadas com uma mesma indentação. Determina-se o início e o fim de um bloco, no Python, através de sua **indentação** (que é o número de espaços no início da linha).

Um bloco inicia quando uma linha é indentada em 4 espaços. Qualquer linha seguinte, será indentada 4 espaços, para que o bloco continue a ser desenvolvido. Um bloco dentro de outro bloco, possui mais 4 espaços a frente, e assim por diante. O bloco termina quando há uma "regressão" dos espaços (por exemplo, para início de um novo bloco).

Na figura 4 há um exemplo de utilização de blocos. As indentações possuem pontos indicando a quantidade de espaços que caracterizam um bloco.

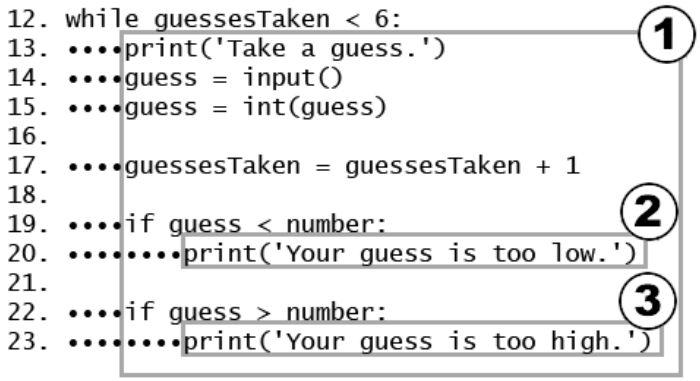

Figura 4: Blocos e indentação.

Ainda na figura 4, a linha 12 possui indentação zero e não está dentro de nenhum bloco. A linha 13 possui 4 espaços no início. A indentação maior que a anterior caracteriza o início de um bloco. As linhas 14, 15, 17, e 19 também possuem a mesma indentação, o que caracteriza que elas pertencem a um mesmo bloco[7].

A linha 20 possui 8 espaços (que são mais que quatro!), logo, um novo bloco foi iniciado. Este bloco está dentro de um outro bloco. A linha 22 possui apenas 4 espaços e a linha anterior possui 8. Isto que dizer que o bloco anterior foi finalizado. O bloco da linha 22 pertence ao mesmo bloco em que as linhas possuem 4 espaços iniciais. A linha 23 aumenta o número de espaços, logo, deu início a um novo bloco.

[6] Embora a estrutura dos *loops* seja bem simples, eventualmente, no Python, eles são tratados um pouco diferentes do que em outras linguagens, por isso vamos abordar alguns tópicos comuns com mais detalhes ao longo do curso.

[7] As linhas em branco não contam!

4.10 Dado booleano

O tipo de dado booleano possui apenas dois valores: `True` (verdadeiro) ou `False` (falso). Estes valores são *case-sensitive* e não são string; em outras palavras, não são utilizadas aspas ao se referir a tais valores. Utilizamos estes valores com operdores de comparação, para formar condições.

4.11 Operadores de comparação

Tabela 3: Operadores de comparação.

Sinal	Nome
<	menor que
>	maior que
<=	menor ou igual a
>=	maior ou igual a
==	igual a
!=	diferente de

Na linha 12 do programa, há uma sentença `while`:

```
12 while guessesTaken < 6:
```

A expressão que segue a palavra-chave `while` contém dois valores (o valor da variável `guessesTaken` e o inteiro `6`) conectados por um operador (o sinal de "menor que" <). O sinal < é chamado de **operador de comparação**.

Este tipo de operador é utilizado para comparar dois valores e concluir se o resultado é `True` ou `False`. Observe os tipos de comparadores disponíveis na tabela 3.

4.12 Condições

Uma **condição** é uma expressão que combina dois valores com um operador de comparação e resulta em um valor booleano. Uma condição é apenas um nome para uma expressão que resulta em `True` ou `False`. Por exemplo, a expressão `guessesTaken < 6` é o mesmo que perguntarmos "o valor armazenado em `guessesTaken` é menor que 6?" Em caso afirmativo, a condição resulta em `True`. Caso contrário, em `False`.

Tarefa 4.1
Entre com as seguintes operações no terminal interativo:

1. $0 < 6$
2. $6 < 0$
3. $50 < 10$
4. $10 < 11$
5. $10 < 10$
6. $10 == 10$
7. $10 == 11$
8. $11 == 10$
9. $10 != 10$
10. $10 != 11$
11. 'Hello' == 'Hello'
12. 'Hello' == 'Good bye'
13. 'Hello' == 'HELLO'
14. 'Good bye' != 'Hello'

Qual é o resultado de cada uma delas? Justifique/comente as respostas que mais lhe chamaram a atenção.

Não deixe de reparar a diferença entre os sinais == e =. Quando utilizamos apenas um sinal de igual, estamos atribuindo um valor, enquanto ao utilizarmos dois sinais, queremos comparar dois valores.

4.13 Loops com sentenças `while`

A sentença `while` marca o início de um loop. Eventualmente, em nossos programas, queremos que algo ocorra várias vezes. Quando a execução chega em um bloco `while`, ela verifica a condição em sequência à palavra-chave `while`. Se a condição resultar em `True`, o bloco é executado. Se a condição resulta em `False`, a execução "pula" o bloco `while` e continua executando o que vier depois.

Um bloco `while` pode ser traduzido da seguinte forma: "enquanto a condição for verdadeira, continue executando o que estiver neste bloco". Observe a figura 5.

É válido lembrar que algum erro de programação pode gerar loops que não terminam! Por isso, estes blocos exigem maior atenção ao serem implementados, ou o programa poderá não sair do bloco e não terminar a execução.

4.14 As tentativas do jogador

As linhas 13 a 17 pedem ao jogador para adivinhar o número sorteado e, obviamente, permite com que um palpite seja dado como entrada. Este palpite é armazenado na variável `guess` e, posteriormente, este valor é convertido de uma string para um inteiro.

```
13      print('Take a guess.')
14      guess = input()
15      guess = int(guess)
```

4.14.1 Convertendo strings para inteiros

Na linha 15, chamamos uma nova função, chamada `int()`. A função `int()` exige um argumento. A função `input()` retorna uma string, do texto que o usuário digitou. Entretanto, neste programa, precisamos de um inteiro e não de uma string. Desta forma, a função `int()` "transformará" o valor string dado e retornará um inteiro.

Você deve se perguntar o que acontece se o argumento dado à esta função não for um número. Você pode testar isto (e outros inúmeros palpites) no terminal interativo.

No jogo "adivinhe o número", se o jogador digitar algo que não seja um número, a chamada da função `int()` resultará em um erro. Além disso, não podemos fazer comparações matemáticas

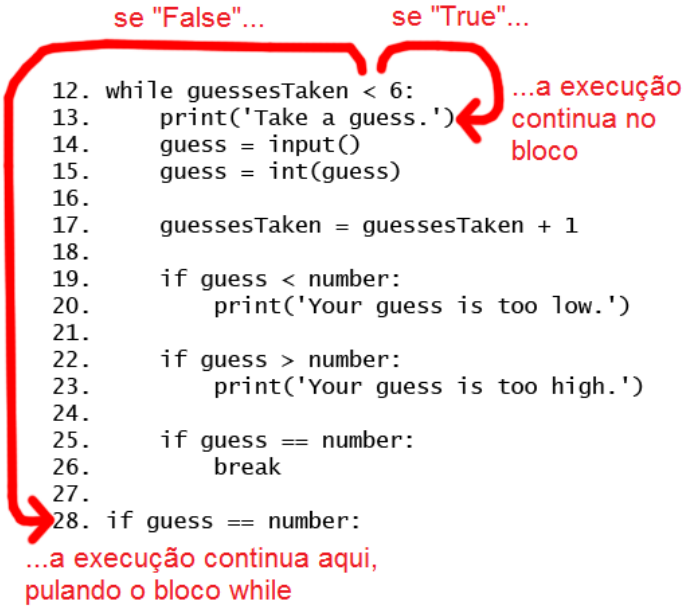

Figura 5: Funcionamento de um *loop* `while`.

entre número e string. Portanto, nos próximos programas, adicionaremos código para condições de erro como esta e dar uma nova chance para o jogador continuar com a execução do programa.

Observe que chamar a função `int(guess)` não modifica o valor da variável `guess`. A expressão `int(guess)` apenas transforma o valor armazenado na variável `guess` em um inteiro. Para que a variável receba o valor inteiro gerado, devemos atribuí-lo novamente à variável. Por isso, a linha 15 se torna útil.

Tarefa 4.2
Verifique o acontece com as seguintes entradas:

1. int('42')
2. int(42)
3. int('hello')
4. int(' 42 ')
5. int('forty-two')
6. 3 + int('2')

Tarefa 4.3
Pesquise sobre a função `int()` e quais os tipos de argumentos que ela aceita.

4.14.2 Incrementando variáveis

```
17     guessesTaken = guessestaken + 1
```

Uma vez que o jogador faz um palpite, devemos aumentar o número de palpites dados, pois existe um limite. Na primeira vez que entrarmos no loop, a variável `guessesTaken` possui o valor zero. O Python pegará este valor e somará 1 à esta variável com a linha 17.

Quando adicionamos 1 a um valor inteiro, usualmente dizemos que estamos **incrementando** alguma variável. Quando subtraímos uma unidade, dizemos que estamos **decrementando** alguma variável.

4.15 Sentenças `if`

```
19      if guess < number:
20          print('Your guess is too low.')
```

A linha 19 inicia uma sentença com a palavra-chave `if`. Seguido do `if`, há uma condição. A linha 20 inicia um novo bloco (devido à indentação). Este é um bloco condicional. Um bloco `if` é utilizado quando queremos executar algo somente quando alguma condição é verdadeira.

Assim como o bloco `while`, o bloco `if` também possui uma palavra-chave, seguida de uma condição, e, então, um bloco de código a ser executado. Observe a figura 6.

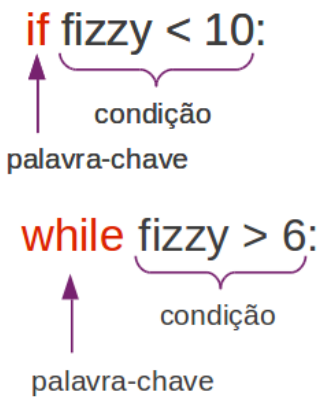

Figura 6: Sentenças `if` e `while`.

Uma sentença `if` funciona da mesma forma que uma sentença `while`. Entretanto, em um bloco `while`, a execução não sai imediatamente após o fim do bloco, mas continua enquanto a condição for verdadeira.

Se a condição é `True`, então, todas as linhas do bloco `if` são executadas. Por exemplo, há uma única linha dentro do bloco `if`, iniciado na linha 19, chamando uma função `print()`. Se o inteiro que o jogador der entrada for menor que o inteiro aleatório gerado, o programa mostra `Your guess is too low`. Se o palpite do jogador for maior que o número gerado, então a condição deste bloco resultará em `False`, e o bloco não será executado.

4.16 Saindo de *loops* com o comando `break`

```
25      if guess == number:
26          break
```

Este bloco `if` verifica se o palpite do jogador é igual ao inteiro gerado no programa. Se for, o programa entra em um bloco que possui uma sentença `break`, que faz com que o programa saia do bloco `while` para a primeira linha seguinte a ele[8].

No caso deste jogo, se o palpite do jogador não for igual ao número gerado, ele continua executando o loop, até que as chances do jogador se esgotem. Se o loop terminar ou for interrompido com o `break`, a próxima ação a ser realizada pelo computador é executar as linhas de código restantes.

4.17 Verificar se o jogador venceu

```
28  if guess == number:
```

Diferentemente do código da linha 25, este bloco `if` não possui indentação. Logo, isto significa que o bloco `while` terminou na linha anterior a este. Para entrarmos neste bloco `if` saimos do bloco

[8] Um comando `break` não faz a verificação da condição do loop novamente, apenas encerra o bloco imediatamente.

while por ter extrapolado as tentativas ou por ter adivinhado o número. Se for o caso de o jogador ter acertado o número, o programa executará o código seguinte à linha 28.

```
28 if guess == number:
29     guessesTaken = str(guessesTaken)
30     print('Good job, ' + myName + '! You guessed my number in ' +
31         guessesTaken + ' guesses!')
```

As linhas 29 e 30 estão dentro do bloco if, que serão executadas apenas se a condição da linha 28 for verdadeira. Na linha 29, chamamos a função str() que retorna uma string através do argumento dado. Esta linha se torna útil, pois para imprimir na tela, chamando a função print(), devem ser utilizados apenas strings, já que só é permitida a operação de concatenação entre strings.

4.17.1 Verificar se o jogador perdeu

```
33 if guess != number:
34     number = str(number)
35     print('Nope. The number I was thinking of was ' + number)
```

Na linha 33, utilizamos o operador de comparação $!=$ na sentença if, significando "diferente de". Se as chances do jogador se esgotarem, então a condição deste bloco será verdadeira, e as linhas 34 e 35 serão executadas. Desta forma, é dito ao jogador que ele falhou em adivinhar o número, além de informá-lo qual era este número, de forma análoga ao bloco if anterior.

4.18 Resumo: o que exatamente é programação?

Se alguém lhe perguntar "o que exatamente é programação?", qual seria a sua resposta? Programar é apenas escrever códigos para programas que podem ser executados por um computador.

"Mas, exatamente, o que é um programa?" Quando você vê alguém utilizando um programa, como este que acabamos de estudar, tudo o que vemos é um texto aparecendo na tela. O programa decide o que será mostrado na tela (chamado de **output**), baseado em suas instruções e no texto dado como entrada pelo jogador (ou seja, o **input**). O programa apenas possui instruções sobre o que mostrar ao usuário. Ou seja, um **programa** nada mais é que uma coleção de instruções.

"Que tipos de instruções?" Há alguns tipos diferentes de instruções, na verdade. Expressões são instruções que resultam em um valor final, como $2+2$ resulta em 4. Chamadas de função também são partes de expressões, pois elas resultam em um valor único, que pode ser conectado a outros valores por meio de operadores. Quando expressões estão acompanhadas de palavras-chave como if ou while, elas são chamadas de condições. Sentenças de atribuição servem para guardar valores em variáveis para que estes possam ser utilizados posteriormente.

if, while e break são exemplos de sentenças de **controle de fluxo**, pois elas estabelecem quais instruções são executadas. O fluxo normal de execução para um programa é iniciar da primeira linha e executar cada instrução, uma a uma. Entretanto, estas estruturas permitem que instruções sejam puladas, repetidas ou interrompidas. Chamadas de função também mudam o fluxo de execução, pulando para onde a função se encontra.

A função print() mostra um texto na tela. A função input() armazena um texto dado como entrada, pelo teclado. Isto é chamado de operação de **I/O** (input/output), pois lida com entradas e saídas do programa.

Por enquanto, temos tudo isto como início do trabalho. Com o passar do tempo, serão aprendidos novos conceitos sobre tipos de dados e operadores, além de sentenças de controle de fluxo e novas funções. Também veremos diferentes tipos de I/O (entrada pelo mouse; saída de sons, gráficos e imagens, etc.).

Para o usuário, apenas importam as operações de I/O de um programa. O jogador digita alguma coisa ou clica em outra e quer ver o que acontece, através de sons e imagens. Mas, para o computador produzir qualquer saída é necessário um programa, uma série de instruções que iremos desenvolver.

4.19 Exercícios complementares

1. Faça um Programa que leia três números e mostre o maior deles.
2. Faça um Programa que verifique se uma letra digitada é "F" ou "M". Conforme a letra escrever: F - Feminino, M - Masculino, Sexo Inválido.
3. Tendo como dados de entrada a altura e o sexo e peso de uma pessoa, construa um algoritmo que calcule seu peso ideal e informe se ela está dentro, acima ou abaixo do peso, utilizando as seguintes fórmulas: Para homens: $(72.7 * h) - 58$; Para mulheres: $(62.1 * h) - 44.7$ (h = altura)
4. Faça um Programa que pergunte em que turno você estuda. Peça para digitar M-matutino ou V-Vespertino ou N- Noturno. Imprima a mensagem "Bom Dia!", "Boa Tarde!" ou "Boa Noite!" ou "Valor Inválido!", conforme o caso.
5. Faça um programa que faça 5 perguntas para uma pessoa sobre um crime. As perguntas são: "Telefonou para a vítima?", "Esteve no local do crime?", "Mora perto da vítima?", "Devia para a vítima?" e "Já trabalhou com a vítima?" O programa deve no final emitir uma classificação sobre a participação da pessoa no crime. Se a pessoa responder positivamente a 2 questões ela deve ser classificada como "Suspeita", entre 3 e 4 como "Cúmplice" e 5 como "Assassino". Caso contrário, ele será classificado como "Inocente". Dica: use uma variável contadora responsável por contar cada vez que o usuário responda "sim"
6. Faça um programa que peça uma nota, entre zero e dez. Mostre uma mensagem caso o valor seja inválido e continue pedindo até que o usuário informe um valor válido. Dica: use o comando `while`.
7. Faça um programa que leia um nome de usuário e a sua senha e não aceite a senha igual ao nome do usuário, mostrando uma mensagem de erro e voltando a pedir as informações.

5 O Reino do Dragão

Tópicos abordados neste capítulo:

- O módulo `time`
- A função `time.sleep()`
- A palavra-chave `return`
- Criar as próprias funções com a palavra-chave `def`
- Os operadores booleanos `and`, `or` e `not`
- Tabelas-verdade
- Escopo de variável local e global
- Parâmetros e argumentos
- Diagramas de execução

5.1 Introdução à funções

Já utilizamos duas funções, anteriormente, `input()` e `print()`. Nos programas anteriores, chamamos estas funções para que o código dentro delas fosse executado. Neste capítulo, faremos as nossas próprias funções. Conforme já mencionado, uma função é como um programa dentro do nosso programa. Muitas vezes, queremos que o nosso programa execute um mesmo procedimento várias vezes, em diferentes pontos do código, de forma que não podemos fazer um simples loop. Ao invés de digitá-lo inúmeras vezes, construimos uma função e a chamamos sempre que quisermos. Este recurso possui a vantagem de, caso haja algum erro, fazermos correções em um único bloco de código.

O programa que construiremos para aprender funções é chamado "O Reino do Dragão" e permite que o jogador faça um palpite entre duas cavernas que, aleatoriamente, possuem um tesouro ou maldição.

5.2 Como jogar "O Reino do Dragão"

Neste jogo, o jogador está em uma terra cheia de dragões. Os dragões vivem em cavernas com grandes quantidades de tesouros. Alguns dragões são amigáveis e compartilharão o tesouro com o jogador. Outros são gananciosos e famintos, e devorarão qualquer um que entrar em sua caverna. O jogador está em frente a duas cavernas, uma com um dragão amigável e outra com um ganancioso. Ele pode escolher entre uma das cavernas.

5.2.1 Modelo de saída do jogo

```
You are in a land full of dragons. In front of you,
you see two caves. In one cave, the dragon is friendly
and will share his treasure with you. The other dragon
is greedy and hungry, and will eat you on sight.
Which cave will you go into? (1 or 2)
1
You approach the cave...
It is dark and spooky...
A large dragon jumps out in front of you! He opens his jaws
and...
Gobbles you down in one bite!
Do you want to play again? (yes or no)
no
```

5.3 Código-fonte do programa

Aqui está o código fonte do jogo. Digitá-lo é uma boa prática para tornar-se acostumado com o código.

O principal a saber sobre o código 3: os blocos precedentes por `def` são como as funções são declaradas no Python. Entretanto, o código em cada bloco de função não é executado até que a função seja chamada. Desta forma, o programa não será executado linha a linha, da primeira a última, conforme seria o esperado. Isto será discutido mais adiante.

Código 3: O Reino do Dragão.

```
1  import random
2  import time
3
4  def displayIntro():
5      print('You are in a land full of dragons. In front of you,')
6      print('you see two caves. In one cave, the dragon is friendly')
7      print('and will share his treasure with you. The other dragon')
8      print('is greedy and hungry, and will eat you on sight.')
9      print()
10
11 def chooseCave():
12     cave = ''
13     while cave != '1' and cave != '2':
14         print('Which cave will you go into? (1 or 2)')
15         cave = input()
16
17     return cave
18
19 def checkCave(chosenCave):
20     print('You approach the cave...')
21     time.sleep(2)
22     print('It is dark and spooky...')
23     time.sleep(2)
24     print('A large dragon jumps out in front of you! He opens his jaws and...
           ')
25     print()
26     time.sleep(2)
27
28     friendlyCave = random.randint(1, 2)
29
30     if chosenCave == str(friendlyCave):
31          print('Gives you his treasure!')
32     else:
33          print('Gobbles you down in one bite!')
34
35 playAgain = 'yes'
36 while playAgain == 'yes' or playAgain == 'y':
37
38     displayIntro()
39
40     caveNumber = chooseCave()
41
42     checkCave(caveNumber)
43
44     print('Do you want to play again? (yes or no)')
45     playAgain = input()
```

5.4 Como o programa funciona

Aqui, temos duas sentenças de importação. Importamos o módulo `random`, como no jogo "adivinhe o número". Neste programa, também precisaremos de funções relacionadas a tempo, por isso importamos o módulo `time`.

5.4.1 Definindo funções

```
4  def displayIntro():
5      print('You are in a land full of dragons. In front of you,')
6      print('you see two caves. In one cave, the dragon is friendly')
7      print('and will share his treasure with you. The other dragon')
8      print('is greedy and hungry, and will eat you on sight.')
9      print()
```

A figura 7 mostra um novo tipo de sentença, uma declaração de função. Este tipo de bloco é feito com a palavra-chave `def`, seguido do nome da função e um par de parênteses. Para finalizar, dois pontos para abrir um bloco.

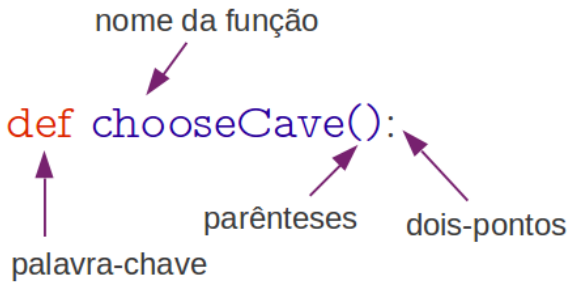

Figura 7: Partes de uma definição de função no Python.

O bloco que inicia na linha 4 não é uma chamada para a função `displayIntro()`. Ao invés disso, uma função está sendo criada (ou definida), para que possamos chamá-la posteriormente no programa. Depois de **definirmos**[9] esta função, podemos chamá-la da mesma forma que fizemos com outras funções, assim, o código dentro dela será executado.

Assim que chamarmos a função, a execução pula para a linha 5 do programa. Quando a execução da função termina, o programa continua na linha seguinte em que ela foi chamada.

Na função `chooseCave()`, temos as linhas:

```
12     cave = ''
13     while cave != '1' and cave != '2':
```

em que criamos uma variável chamada `cave` e armazenamos uma string vazia. Então, iniciamos um loop `while`. Neste loop, há um novo operador, que ainda não foi abordado, o operador `and`. Assim como o sinal de menos ($-$) ou de multiplicação ($*$) são operadores matemáticos, e $==$ ou $!=$ são operadores de comparação, o `and` é um operador booleano.

5.4.2 Operadores booleanos

A lógica booleana lida com operações que resultam em verdadeiro ou falso. Operadores booleanos comparam dois valores e resultam em um único valor booleano. Por exemplo, o operador `and` avalia se uma expressão é verdadeira quando ambos os lados forem verdadeiros. Observe: "gatos miam e cachorros latem". Esta sentença é verdadeira, pois ambos os lados (esquecendo de eventuais exceções) são verdadeiros. Para uma sentença `and` ser verdadeira, vale a mesma analogia. A sentença inteira será verdadeira se, e somente se, ambos os lados forem verdadeiros.

Agora, voltando à linha 13:

[9] Utilizamos o termo "definir" para variáveis também. Assim como "declarar".

```
13      while cave != '1' and cave != '2':
```

A condição que acompanha o bloco `while` possui duas expressões conectadas pelo operador `and`. Primeiro, cada expressão é avaliada como `True` ou `False` para depois a expressão inteira resultar também em um valor booleano.

A string armazenada na variável `cave` é vazia, logo, é diferente de `'1'` ou `'2'`, assim, todas as expressões são verdadeiras. Desta forma, entramos no loop, pois a expressão resulta em `True`.

Além do `and`, existem outros operadores booleanos, como o `or` e o `not`. No caso do operador `or`, uma sentença é verdadeira se *pelo menos* uma das expressões é verdadeira. O operador `not` é análogo ao `!=`, visto anteriormente, entretanto, ele trabalha apenas com um valor, invertendo o valor booleano de alguma entidade. Por exemplo, observe e teste estas sentenças no terminal interativo:

```
>>> not True
False
>>> not False
True
>>> True and not False
True
```

5.4.3 Tabelas-verdade

Para observar os resultados de operadores booleanos, frequentemente é utilizado um recurso chamado tabela-verdade. Uma tabela-verdade mostra todas as possibilidades de uma expressão booleana de acordo com os possíveis valores de cada variável. Observe as tabelas 4, 5 e 6.

Tabela 4: Tabela-verdade do operador `and`

A	and	B	Resultado
True	and	True	True
True	and	False	False
False	and	True	False
False	and	False	False

Tabela 5: Tabela-verdade do operador `or`

A	or	B	Resultado
True	or	True	True
True	or	False	True
False	or	True	True
False	or	False	False

Tabela 6: Tabela-verdade do operador `not`

not	A	Resultado
not	True	False
not	False	True

5.4.4 Obtendo a entrada do jogador

```
14          print('Which cave will you go into? (1 or 2)')
15          cave = input()
```

Neste trecho, é pedido ao jogador para digitar 1 ou 2 e pressionar Enter. Qualquer string que o jogador digitar será armazenada em `cave`. Mas, se o jogador digitar 1 ou 2, então o valor de `cave` será `'1'` ou `'2'`. Se for este o caso, então a condição do loop será falsa e o programa não entrará mais no loop.

Em outras palavras, o motivo pela existência deste loop é no caso de o jogador digitar qualquer outra coisa além de 1 ou 2. O computador perguntará várias vezes a mesma coisa, até que o jogador dê alguma resposta válida.

5.4.5 Retorno de valores

```
17     return cave
```

Uma sentença de retorno apenas aparece em blocos de função. Lembre-se que a função `input()` retornava qualquer valor que fosse dado como entrada pelo teclado. Na função `chooseCave()`, o que será retornado é o valor que estiver armazenado na variável `cave`.

Isto significa que se tivéssemos alguma sentença do tipo `spam = chooseCave()`, o código dentro da função `chooseCave()` será executado e o resultado será armazenado na variável `spam`. Neste caso, o valor de retorno (ou seja, que seria armazenado na variável `spam`) poderia assumir os valores `'1'` ou `'2'`, pois o loop `while` garante que, ao chegar na linha 17, o valor de `cave` seja apenas um deles.

Uma sentença com `return` faz o programa sair imediatamente de uma função, assim como o `break` interrompe um loop. A execução volta à linha seguinte em que a função foi chamada. Também é possível usar apenas `return` para sair de uma função, sem atribuir nenhum valor de retorno.

5.4.6 Escopo de variáveis

Assim como os valores armazenados em variáveis são esquecidos quando um programa termina, variáveis criadas dentro de funções são esquecidas assim que estas terminam a execução. Não apenas isso, mas não é possível alterar variáveis dentro de funções em outros pontos do programa que não seja dentro da função. Isto é definido pelo **escopo** da variável. As variáveis utilizadas dentro de uma função são apenas aquelas que foram criadas dentro de tal função. Variáveis criadas fora das funções, em um programa, possuem escopo (ou acesso) fora das funções.

Por exemplo, se uma variável `spam` foi criada dentro de um função, e outra, com o mesmo nome, foi criada fora da função, o Python interpretará como se fossem duas variáveis distintas. Isto significa que, se alterarmos o valor de `spam` dentro da função, o valor da variável com o mesmo nome, fora da função, não será alterado. Isto é porque as variáveis possuem escopo diferente, global e local.

Escopo global e local

O escopo que abrange a área fora de qualquer função é chamado de **escopo global**. Dentro de uma função, chama-se **escopo local**. Cada programa possui apenas um escopo global e cada função possui um escopo local próprio.

5.4.7 Definindo a função `checkCave(chosenCave)`

```
19 def checkCave(chosenCave):
```

Observa-se que, ao declarar a função `checkCave()`, há um texto dentro dos parênteses da função, no caso, `chosenCave`. Este "texto" dentro dos parênteses são variáveis, chamadas **parâmetros**.

Lembre-se que já passamos parâmetros para funções anteriormente, com as funções `str()` e `randint()`. Ao chamarmos `checkCave()`, devemos passar um valor como parâmetro. Este valor é identificado dentro da função com o nome de `chosenCave`. Esta é a forma com que passamos valores para funções, desde que elas não possuem acesso a variáveis fora de seu escopo.

Em resumo, parâmetros são variáveis locais que são definidas quando a função é chamada. O valor armazenado no parâmetro é o argumento que foi passado na chamada da função.

5.4.8 Onde definir funções

Uma definição de função, em Python, deve vir antes que ela seja chamada. É a mesma ideia de ter que atribuir um valor a uma variável antes que ela seja utilizada. Observe:

```
1 sayGoodBye()
2
3 def sayGoodBye():
4     print('Good bye!')
```

Este código gera um erro, pois a função foi chamada antes de ser definida. Para corrigir isto, devemos apenas mover a chamada da função para depois da definição da mesma.

```
1 def sayGoodBye():
2     print('Good bye!')
3
4 sayGoodBye()
```

5.4.9 Mostrando o resultado do jogo

De volta ao código do programa:

```
20     print('You approach the cave...')
21     time.sleep(2)
22     print('It is dark and spooky...')
23     time.sleep(2)
24     print('A large dragon jumps out in front of you! He opens his jaws and...')
25     print()
26     time.sleep(2)
```

Mostramos um texto ao jogador e depois chamamos a função `time.sleep()`. Lembre-se de como foi chamada a função `randint()` no jogo de adivinhar o número. A função `sleep()` faz parte do módulo `time`, importado no início do programa. Esta função faz com que o programa dê uma pausa de quantos segundos lhe forem dados como argumento. Desta forma, a cada vez em que a função `sleep()` é chamada neste programa, é dada uma pausa de dois segundos.

Estas pausas dão suspense ao jogo, ao invés de mostrar o todo o texto de uma vez.

5.4.10 Decidindo em que caverna está o dragão amigável

```
28     friendlyCave = random.randint(1, 2)
29
30     if chosenCave == str(friendlyCave):
31         print('Gives you his treasure!')
32     else:
33         print('Gobbles you down in one bite!')
```

Agora o programa selecionará aleatoriamente em que caverna o dragão amigável está. Para isto, chamamos a função `random.randint()` com um retorno entre os números 1 e 2 e o armazenamos na variável `friendlyCave`. Em seguida verificamos se o inteiro gerado é igual à entrada do usuário, transformando o número em uma string[10].

Também poderia ser feito o contrário, ou seja, transformar a string digitada pelo usuário em um inteiro. Observe:

```
if int(chosenCave) == friendlyCave:
```

Finalmente, se a condição do bloco `if` for verdadeira, o jogador é informado de que ele ganhou o jogo. Do contrário, o bloco `else` é executado e o jogador perde o jogo.

[10] Lembre-se que só podemos fazer comparações de uma string com outra string, por isso utilizamos a função `str()`, já que o retorno da função `randint()` é um inteiro. Ao comparar valores de diferentes tipo, o a operação de igualdade `==` sempre resultará em `False`.

Dois-pontos

Você pode ter notado que os blocos `if`, `else`, `while` ou função, têm sua primeira linha seguida por dois pontos. Este sinal marca o fim de uma sentença e nos informa que a próxima linha é o início de um novo bloco.

5.4.11 Onde o programa realmente começa

```
35 playAgain = 'yes'
36 while playAgain == 'yes' or playAgain == 'y':
37
38     displayIntro()
39
40     caveNumber = chooseCave()
41
42     checkCave(caveNumber)
43
44     print('Do you want to play again? (yes or no)')
45     playAgain = input()
```

A linha 35 é a primeira linha que não é uma função (ou seja, que não está dentro de um bloco `def`). É nesta linha que o programa realmente começa. Os blocos anteriores são apenas declaração de funções que não são executadas se não forem chamadas.

A linha 36 sinaliza o início de um loop `while`. Entramos a primeira vez no loop através do valor da variável `playAgain`, que foi iniciada com `'yes'` e faz com que a condição seja verdadeira.

5.4.12 Chamando funções no programa

Na linha 38, chamamos a função `displayIntro()`. Esta não é uma função do Python, mas uma função que criamos. Quando esta função é chamada, o programa pula para a linha 5 e executa o que está no bloco da respectiva função. Quando todas as linhas da função forem executadas, a execução continua na linha seguinte em que a função foi chamada.

Em seguida, uma variável recebe o valor gerado pela função `chooseCave()`, que também foi criada neste programa, para o jogador escolher qual caverna seguir. Feito isso, a execução continua e envia o valor da variável `caveNumber` à função `checkCave()`. Esta função vai comparar os valores do jogador e da caverna e mostrará na tela o resultado do jogo, dependendo de onde o jogador escolheu ir.

5.4.13 Perguntando se o jogador quer jogar novamente

Depois que o jogo terminar, o jogador tem a escolha de jogar novamente, antes de finalizar o programa. A variável `playAgain` armazena a escolha do jogador. Entretanto, diferente da função `chooseCave()`, discutida anteriormente, o loop final só será executado se o jogador digitar `'yes'` ou `'y'`[11]. Se o jogador digitar qualquer outra coisa, a condição resultará em `False` e o loop é pulado. Contudo, não há mais linhas de código a serem executadas, após este loop, e o programa, então, termina.

5.5 Projetando um programa

O Reino do Dragão é um jogo bem simples. Os próximos serão um pouco mais complicados. Desta forma, é de grande ajuda escrever (sim, no papel) o será feito em um programa. Isto se chama "projetar um programa".

Por exemplo, desenhar um **fluxograma** pode ajudar no entendimento. Um fluxograma é um desenho que mostra cada ação possível que deve acontecer no programa e em que ordem elas devem ocorrer. Normalmente, criamos estes projetos antes de escrever um programa, pois assim lembramos de escrever o código de todas as partes do programa. Observe a figura 8.

[11] Nem se o jogador digitar `'YES'` a condição resultará em `True`, pois, `'yes'` é diferente de `'YES'`, para o Python. Verifique você mesmo no terminal interativo.

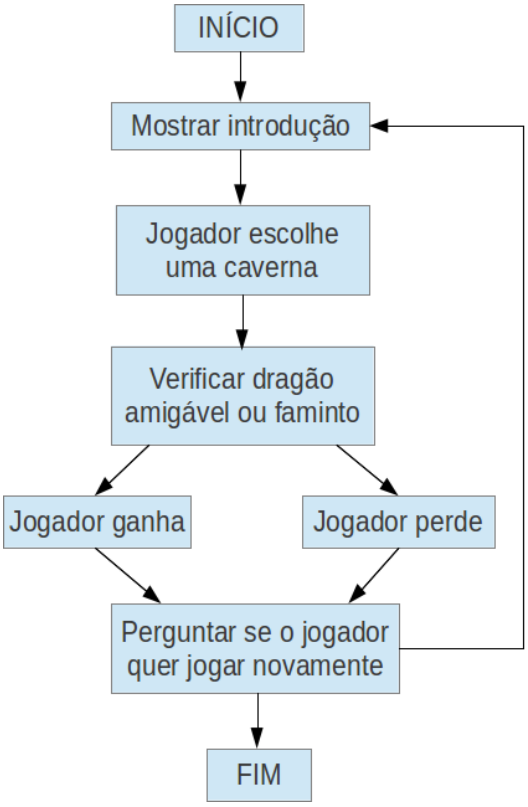

Figura 8: Fluxograma para o jogo "O Reino do Dragão".

Para entender o que acontece no jogo, siga as setas a partir do quadrado que diz "INÍCIO". Perceba que, ao chegar no quadrado "Verificar dragão amigável ou faminto" o programa escolherá se o jogador ganha ou perde. Então, o programa pergunta se o jogador deseja jogar novamente e ele é executado novamente ou termina.

5.6 Resumo

No jogo "O Reino do Dragão", criamos nossas próprias funções. Podemos pensar em funções como mini-programas dentro do nosso programa, que só serão executados quando a função for chamada. Utilizar funções pode tornar o código mais simples para ser lido. Também podemos executar várias vezes um mesmo código utilizando funções.

As entradas para funções são os *argumentos* passados em uma chamada de função. A função envia ao programa principal uma resposta, ou seja, um valor de retorno, que é a saída da função. Há ainda funções que não retornam nada.

Também aprendemos sobre escopo de variáveis. Variáveis criadas dentro de uma função possuem escopo local, enquanto aquelas criadas fora de funções são de escopo global. O código em escopo global não tem acesso à variáveis locais. Se uma variável local possui o mesmo nome de uma variável global, o Python irá interpretá-las como duas variáveis distintas. Em outras palavras, alterar o valor de uma delas não altera o valor da outra.

Entender como o escopo de variáveis funciona pode parecer simples, mas ao utilizá-lo pode parecer complicado. Entretanto, é uma ferramenta útil para organizar funções como pedaços de código independentes em um programa. Como cada função tem seu escopo local, podemos ter certeza de que uma função não interferirá em outra parte do código.

Programas mais próximos de um nível profissional utilizam funções, assim como o restante dos jogos que serão vistos. Ao entender como as funções funcionam, podemos facilitar o trabalho futuramente, reaproveitando funções já construídas.

Tarefa 5.1

1. Utilize o código 3 e faça um jogo mais elaborado, com os conceitos aprendidos até agora.
2. Pesquise sobre o *debugger* do IDLE.

5.7 Exercícios complementares

1. Faça um programa que converta da notação de 24 horas para a notação de 12 horas. Por exemplo, o programa deve converter 14:25 em 2:25 P.M. A entrada é dada em dois inteiros. Deve haver pelo menos duas funções: uma para fazer a conversão e uma para a saída. Registre a informação A.M./P.M. como um valor 'A' para A.M. e 'P' para P.M. Assim, a função para efetuar as conversões terá um parâmetro formal para registrar se é A.M.
2. **Jogo de Craps**. Faça um programa de implemente um jogo de Craps. O jogador lança um par de dados, obtendo um valor entre 2 e 12. Se, na primeira jogada, você tirar 7 ou 11, você é um "natural" e ganhou. Se você tirar 2, 3 ou 12 na primeira jogada, isto é chamado de "craps" e você perdeu. Se, na primeira jogada, você fez um 4, 5, 6, 8, 9 ou 10, este é seu "Ponto". Seu objetivo agora é continuar jogando os dados até tirar este número novamente. Você perde, no entanto, se tirar um 7 antes de tirar este Ponto novamente.
3. **Data com mês por extenso**. Construa uma função que receba uma data no formato $DD/MM/AAAA$ e devolva uma string no formato D de `mesPorExtenso` de AAAA. Opcionalmente, valide a data e retorne NULL caso a data seja inválida.
4. **Desenha moldura**. Construa uma função que desenhe um retângulo usando os caracteres '+' , '−' e '|'. Esta função deve receber dois parâmetros, linhas e colunas, sendo que o valor por omissão é o valor mínimo igual a 1 e o valor máximo é 20. Se valores fora da faixa forem informados, eles devem ser modificados para valores dentro da faixa de forma elegante.

6 Jogo da forca

Tópicos abordados neste capítulo:

- Projeto de um programa
- Arte ASCII
- Métodos
- O método de lista `append()`
- Os métodos de string `lower()` e `upper()`
- O método de lista `reverse()`
- O método de string `split()`
- A função `range()`
- A função `list()`
- Loops `for`
- Sentenças `elif`
- Os métodos de string `startswith()` e `endswith()`

Neste capítulo, será desenvolvido um jogo de forca. Este jogo é mais complicado do que os anteriores, mas também é muito mais divertido. Devido à maior complexidade do programa, vamos inicialmente projetá-lo, utilizando um fluxograma.

Este jogo apresenta vários conceitos novos. Para nos acostumarmos com tais conceitos, o terminal interativo é de grande ajuda. Alguns tipos de dados (como strings ou listas) possuem funções associadas a seus valores, chamadas métodos. Veremos alguns destes métodos, além de uma nova forma de loop (`for`) e blocos condicionais mais elaborados.

6.1 Modelo de saída do jogo de forca

Aqui está um exemplo do que o jogador verá quando executar o jogo de forca.

```
H A N G M A N

  +---+
  |   |
      |
      |
      |
      |
=========
Missed letters:
_ _ _
Guess a letter.
a

  +---+
  |   |
      |
      |
      |
      |
=========
Missed letters:
_ a _
Guess a letter.
o

  +---+
  |   |
  O   |
      |
      |
      |
=========
Missed letters: o
_ a _
Guess a letter.
r

  +---+
  |   |
  O   |
      |
      |
      |
=========
Missed letters: or
_ a _
Guess a letter.
t




  +---+
  |   |
  O   |
      |
      |
```

```
      |
=========
Missed letters: or
_ a t
Guess a letter.
a
You have already guessed that letter. Choose again.
Guess a letter.
c
Yes! The secret word is "cat"! You have won!
Do you want to play again? (yes or no)
no
```

6.2 Arte ASCII

Como pode-se perceber no exemplo de execução do jogo da forca, algumas linhas de saída não são bem um código (com comandos, etc), mas strings de várias linhas que utilizam caracteres do teclado para desenhar imagens. Este tipo de recurso gráfico é chamado de **arte ASCII**, pois os caracteres do teclado são chamados de caracteres ASCII[12]. Para as primeiras aplicações, em que não estamos lidando com imagens e sons, uma saída de texto criativa pode chamar a atenção do jogador.

Observe o exemplo disponível no site http://www.chris.com/ascii/:

```
                                    _
                                   (_)
              |    .
          .   |L  /|   .           _
      _ . |\ _| \--+._/| .        (_)
     / ||\| Y J  )   / |/| ./
    J  |)'( |        ` F`.'/         _
  -<|  F         __      .-<        (_)
    | /       .-'. `.  /-. L___
    J \       <    \  | | O\|.-'  _
  _J \  .-    \/ O | | \  |F     (_)
 '-F  -<_.     \   .-'  `-' L__
__J  _   _.     >-'  ).._.   |-'
`-|.'   /_.           \_|   F
  /.-   .                _.<
 /'    /.'             .'  `\
  /L  /'   |/      _.-'-\
 /'J       ___.---'\|
   |\  .--' V  | `. `
   |/`. `-.     `._)
      / .-.\
VK    \ (  `\
       `.\
```

E o exemplo disponível em http://www.asciiarte.com/index.html.

```
     ___,A.A_  __
     \   ,   7"_/
      ~"T(  r r)
         | \    Y
         |  ~\ .|
         |   |`-'
         |   |
         |   |
         |   |
         |   |
         j   l
        /     \
       Y       Y
       l   \ : |
       /\   )( (
      /  \  I| |\
```

[12] American Standard Code for Information Interchange.

```
    Y     I l| | Y
    j     ( )( ) l
   / .     \ ` | |\    -Row
  Y   \     i   | ! Y
  l    .\_  I   |/  |
   \ /    [\[][]/] j
~~~~~~~~~~~~~~~~~~~~~~
```

6.3 Projetando um programa através de um fluxograma

Este jogo é mais complicado do que já vimos até agora. Então, vamos pensar um pouco sobre o que precisamos e depois partimos para a implementação.

Primeiro, vamos criar um fluxograma, como aquele visto na figura 8, para nos ajudar a visualizar o que este programa fará. Um **fluxograma**[13] é um diagrama que nos mostra uma série de passos, representados por caixas conectadas por setas. Cada caixa representa um passo, enquanto as setas mostram quais passos seguir.

Iniciamos na caixa de início. Vamos percorrendo o fluxograma de acordo com a direção das setas e não podemos voltar algum passo, a não ser que haja alguma seta que nos permita isso. Observe o fluxograma para o jogo da forca na figura 9.

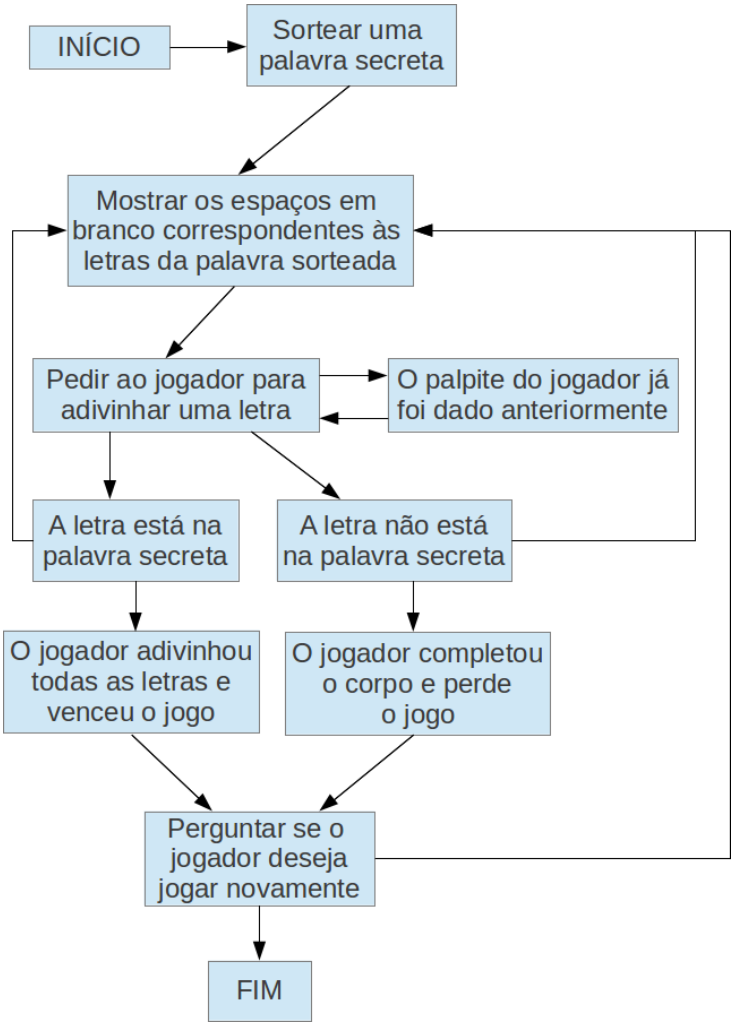

Figura 9: Fluxograma para o jogo da forca.

[13] Há algumas convenções para confecção de fluxogramas, entretanto, não abordaremos tais formalismos. O uso deste recurso será meramente para esclarecer como um programa funciona, mostrando o passo-a-passo de forma que evite confusões na implementação.

É claro que não precisamos fazer um fluxograma e podemos simplesmente começar a escrever o código. Entretanto, ao programar, eventualmente pensamos nas coisas que precisam ser mudadas ou adicionadas que, em um primeiro momento, não tínhamos pensado. Acabamos apagando várias linhas de código e reescrevendo-as, deixando coisas passarem desapercebidas, enfim, coisas que podem ser desperdício de esforço. Para evitar estes desconfortos, é sempre interessante pensar cuidadosamente e planejar o trabalho.

O fluxograma da figura 9 é um exemplo mais robusto de como estes diagramas funcionam. Para trabalhar neste material, não será necessário planejar cada programa, pois eles já estão implementados. Entretanto, ao fazer projetos próprios, esta ferramenta é muito útil e economiza muito tempo de implementação.

6.4 Código-fonte do jogo de forca

Código 4: Jogo de forca.

```
1  import random
2  HANGMANPICS = ['''
3
4     +---+
5     |   |
6         |
7         |
8         |
9         |
10 =========''', '''
11
12    +---+
13    |   |
14    O   |
15        |
16        |
17        |
18 =========''', '''
19
20    +---+
21    |   |
22    O   |
23    |   |
24        |
25        |
26 =========''', '''
27
28    +---+
29    |   |
30    O   |
31   /|   |
32        |
33        |
34 =========''', '''
35
36    +---+
37    |   |
38    O   |
39   /|\  |
40        |
41        |
42 =========''', '''
43
44    +---+
45    |   |
```

```
46      O    |
47     /|\   |
48     /     |
49           |
50    =========''', '''
51
52      +---+
53      |   |
54      O   |
55     /|\  |
56     / \  |
57          |
58    =========''']
59    words = 'ant baboon badger bat bear beaver camel cat clam cobra cougar coyote
          crow deer dog donkey duck eagle ferret fox frog goat goose hawk lion
          lizard llama mole monkey moose mouse mule newt otter owl panda parrot
          pigeon python rabbit ram rat raven rhino salmon seal shark sheep skunk
          sloth snake spider stork swan tiger toad trout turkey turtle weasel whale
          wolf wombat zebra'.split()
60
61    def getRandomWord(wordList):
62        # Esta funcao retorna uma string aleatoria da lista de strings
63        wordIndex = random.randint(0, len(wordList) - 1)
64        return wordList[wordIndex]
65
66    def displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord):
67        print(HANGMANPICS[len(missedLetters)])
68        print()
69
70        print('Missed letters:', end=' ')
71        for letter in missedLetters:
72            print(letter, end=' ')
73        print()
74
75        blanks = '_' * len(secretWord)
76
77        for i in range(len(secretWord)): # substitui os espacos em branco pelas
              letras adivinhadas corretamente
78            if secretWord[i] in correctLetters:
79                blanks = blanks[:i] + secretWord[i] + blanks[i+1:]
80
81        for letter in blanks: # mostra a palavra secreta com espacos entre cada
              letra
82            print(letter, end=' ')
83        print()
84
85    def getGuess(alreadyGuessed):
86        # Retorna a letra que o jogador escolheu. Esta funcao garante que o
              jogador digitou uma unica letra e nada mais.
87        while True:
88            print('Guess a letter.')
89            guess = input()
90            guess = guess.lower()
91            if len(guess) != 1:
92                print('Please enter a single letter.')
93            elif guess in alreadyGuessed:
94                print('You have already guessed that letter. Choose again.')
95            elif guess not in 'abcdefghijklmnopqrstuvwxyz':
96                print('Please enter a LETTER.')
97            else:
98                return guess
99
```

```
100 def playAgain():
101     # Esta funcao retorna True se o jogador quiser jogar novamente. Do
            contrario, retorna False.
102     print('Do you want to play again? (yes or no)')
103     return input().lower().startswith('y')
104
105
106 print('H A N G M A N')
107 missedLetters = ''
108 correctLetters = ''
109 secretWord = getRandomWord(words)
110 gameIsDone = False
111
112 while True:
113     displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)
114
115     # Permite que o jogador digite uma letra
116     guess = getGuess(missedLetters + correctLetters)
117
118     if guess in secretWord:
119         correctLetters = correctLetters + guess
120
121         # Verifica se o jogador venceu
122         foundAllLetters = True
123         for i in range(len(secretWord)):
124             if secretWord[i] not in correctLetters:
125                 foundAllLetters = False
126                 break
127         if foundAllLetters:
128             print('Yes! The secret word is "' + secretWord + '"! You have won
                    !')
129             gameIsDone = True
130     else:
131         missedLetters = missedLetters + guess
132
133         # Verifica se o jogador perdeu
134         if len(missedLetters) == len(HANGMANPICS) - 1:
135             displayBoard(HANGMANPICS, missedLetters, correctLetters,
                        secretWord)
136             print('You have run out of guesses!\nAfter ' + str(len(
                        missedLetters)) + ' missed guesses and ' + str(len(
                        correctLetters)) + ' correct guesses, the word was "' +
                        secretWord + '"')
137             gameIsDone = True
138
139     # Pergunta ao jogador que ele gostaria de jogar novamente (apenas se o
            jogo foi encerrado).
140     if gameIsDone:
141         if playAgain():
142             missedLetters = ''
143             correctLetters = ''
144             gameIsDone = False
145             secretWord = getRandomWord(words)
146         else:
147             break
```

6.5 Como funcionam algumas coisas

Iniciamos importando o módulo `random`, que nos ajudará a escolher aleatoriamente uma palavra secreta da lista de palavras no programa. Em seguida, definimos as strings que ilustrarão o jogo.

Para ajudar no entendimento das estruturas utilizadas no programa, vamos aprender sobre strings de várias linhas e listas.

```
1  import random
2  HANGMANPICS = ['''
3
4    +---+
5    |   |
6        |
7        |
8        |
9        |
10 =========''', '''
11
12 # o resto do bloco é muito grande para ser mostrado aqui
```

A linha 2 é simplesmente uma declaração de variável, mas ela ocupa várias linhas no código-fonte. Esta "linha" vai até a linha 58

6.5.1 Strings de várias linhas

Normalmente, quando você escreve strings em seu código-fonte, elas devem ocupar uma única linha. Entretanto, utilizando três aspas simples para iniciar e terminar uma string, ela pode possuir várias linhas. Por exemplo, verifique o seguinte no terminal interativo:

```
>>> fizz = '''Dear Alice,
I will return home at the end of the month. I will
see you then.
Your friend,
Bob'''
>>> print fizz
```

Se não existisse esta estrutura, teríamos que utilizar o comando especial para pular linha \n. Porém, isto pode fazer com que a string fique mais difícil de ser lida, pelo programador, como no exemplo a seguir.

```
>>> fizz = 'Dear Alice,\nI will return home at the
end of the month. I will see you then.\nYour
friend,\nBob'
>>> print fizz
```

Strings de várias linhas não necessitam ter a mesma indentação para permanecer no mesmo bloco. Para este tipo de string, o Python ignora as regras de indentação.

6.5.2 Constantes

Você deve ter notado que a variável `HANGMANPICS` está com todas as letras em maiúsculas. Esta é a convenção de programação para declarar constantes. **Constantes** são variáveis cujos valores não mudam ao longo do programa. Embora sejamos permitidos a alterar o valor destas variáveis, o programador é alertado a não escrever código que as alterem.

Constantes são muito úteis para fornecer descrições para valores que tenham um significado especial. A variável `HANGMANPICS` nunca muda, e é muito mais fácil escrever `HANGMANPICS` do que escrever cada string. Entretanto, como qualquer convenção, não somos obrigados a usar constantes ou utilizar letras maiúsculas para declará-las. Ainda assim, estas convenções fazem com que o código seja mais fácil de ser lido, principalmente, por outros programadores ou quando você for consultar o código futuramente.

6.5.3 Listas

Agora, aprenderemos sobre um novo tipo de dado, chamado **lista**. Uma lista contém alguns valores armazenados. Por exemplo, `['apples', 'oranges', 'HELLO WORLD']` é uma lista que contém três valores string. Como qualquer outro tipo de dado, é possível armazenar uma lista em uma variável, como `spam = ['apples', 'oranges', 'HELLO WORLD']`.

Listas são um eficiente modo de armazenar diferentes valores em uma única variável. Cada valor dentro de uma lista é chamado de **item**. Cada item pode ser acessado utilizando colchetes e a posição do item na lista.

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[0]
'aardvark'
>>> animals[1]
'anteater'
>>> animals[2]
'antelope'
>>> animals[3]
'albert'
>>>
```

O número entre os colchetes é denominado **índice**. Em Python, o primeiro índice é o número zero, ao invés do número um[14]. Assim, com o primeiro índice sendo o 0, o segundo índice é o 1, o terceiro é o 2, etc. Uma lista é um recurso muito bom quando precisamos armazenar vários valores, sem ter que designar uma variável para cada valor. Do contrário, teríamos o seguinte para o exemplo anterior:

```
>>> animals1 = 'aardvark'
>>> animals2 = 'anteater'
>>> animals3 = 'antelope'
>>> animals4 = 'albert'
```

Utilizando listas, podemos tratar os itens como qualquer outro valor. Observe:

```
>>> animals[0] + animals[2]
'aardvarkantelope'
```

Por `animals[0]` resultar na string `'aardvark'` e `animals[2]` resultar em `'antelope'`, então a expressão `animals[0] + animals[2]` resulta no mesmo que a operação `'aardvark' + 'antelope'`.

Mas o que acontece se tentarmos acessar um índice que é maior do que o número de itens de uma lista? Digite `animals[0]` ou `animals[99]` no terminal interativo.

Alterando valores de itens de uma lista através do uso de índices

É muito simples mudar o valor de um item em uma determinada posição de uma lista. Basta atribuir um novo valor à posição desejada, da mesma forma que isto é feito com qualquer variável. Observe.

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[1] = 'ANTEATER'
>>> animals
['aardvark', 'ANTEATER', 'antelope', 'albert']
```

Concatenando listas

É possível juntar várias listas em uma só com o operador +, da mesma forma que é feito com strings. Quando juntamos listas, a operação é conhecida como **concatenação de listas**. Veja o exemplo.

[14] Esta numeração é bem comum entre as linguagens de programação.

```
>>> [1, 2, 3, 4] + ['apples', 'oranges'] + ['Alice', 'Bob']
[1, 2, 3, 4, 'apples', 'oranges', 'Alice', 'Bob']
>>>
```

Comparação entre listas e strings

Uma string, na verdade, é sequência da caracteres, enquanto que uma lista é uma sequência de itens quisquer[15]. Nos dois casos, concatenação e repetição[16] funcionam de forma logicamente idêntica. Elementos individuais podem ser visualizados, mas somente nas listas é possivel altera-los. Veja os exemplos a seguir:

```
>>> name = "John"
>>> print (name[0])
J
>>> name[0] = "L"
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

```
>>> names = ["John","Mary","Carl"]
>>> print (names[1])
Mary
>>> names[1] = "Joe"
>>> print (names)
['John','Joe','Carl']
```

6.5.4 O operador `in`

O operador `in` torna fácil a verificação de um valor dentro de uma lista. Expressões que usam o operador `in` retornam um valor booleano: `True` se o valor está na lista e `False` se o valor não está na lista.

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> 'antelope' in animals
True
```

A expressão `'antelope' in animals` retorna `True` pois a string `'antelope'` pode ser encontrada na lista. Entretanto, a expressão `'ant' in 'animals'` resultaria em `False`. Mas a expressão `'ant' in ['beetle', 'wasp', 'ant']` resultaria em `True`.

6.5.5 Removendo itens de listas

É possível remover itens de uma lista com uma sentença `del`.

```
>>> spam = [2, 4, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 6, 8, 10]
>>>
```

Note que, ao deletar o item que estava na posição 1, o item que estava na posição 2 tomou lugar do item excluído. Os outros itens também tiveram seus índices atualizados. Você pode deletar várias vezes "o mesmo item".

[15] Em algumas linguagens de programação populares, listas e strings são chamadas de vetores, sendo strings vetor de caracteres

[16] Na operação de repetição, você pode multiplicar alguma lista ou string afim de repetir sua impressão. Teste no terminal interativo o comando: `"olá"*5`

```
>>> spam = [2, 4, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 8, 10]
>>> del spam[1]
>>> spam
[2, 10]
```

Lembre-se que o uso do `del` é uma sentença, não uma função ou operador. Logo, não há nenhum valor de retorno.

6.5.6 Listas de listas

Uma lista é uma tipo de dado que pode conter vários valores como itens. Estes itens também podem ser outras listas. Observe.

```
>>> groceries = ['eggs', 'milk', 'soup', 'apples', 'bread']
>>> chores = ['clean', 'mow the lawn', 'go grocery shopping']
>>> favoritePies = ['apple', 'frumbleberry']
>>> listOfLists = [groceries, chores, favoritePies]
>>> listOfLists
[['eggs', 'milk', 'soup', 'apples', 'bread'],
['clean', 'mow the lawn', 'go grocery shopping'],
['apple', 'frumbleberry']]
>>>
```

Alternativamente, o exemplo anterior pode ser designado da seguinte forma.

```
>>> listOfLists = [['eggs', 'milk', 'soup', 'apples', 'bread'],
    ['clean', 'mow the lawn', 'go grocery shopping'],
    ['apple', 'frumbleberry']]
>>> groceries = listOfLists[0]
>>> chores = listOfLists[1]
>>> favoritePies = listOfLists[2]
>>> groceries
['eggs', 'milk', 'soup', 'apples', 'bread']
>>> chores
['clean', 'mow the lawn', 'go grocery shopping']
>>> favoritePies
['apple', 'frumbleberry']
>>>
```

Para acessar os itens das listas dentro de uma lista, é necessário utilizar dois índices. Um para referenciar a sublista e outra para referenciar o item da sublista, como `listOfLists[1][2]` resulta em `'go grocery shopping'`. Isto é porque `listOfLists[1]` resulta na lista `['clean','mow the lawn', 'go grocery shopping'][2]`. Esta, finalmente resulta no terceiro item da lista, `'go grocery shopping'`.

Para tornar mais fácil a identificação dos itens em uma lista de listas, observe a figura 10.

6.5.7 Métodos

Métodos são como funções, mas estão sempre anexados a algum valor. Por exemplo, todos os valores string possuem um método `lower()`, que retorna uma cópia da string em letras minúsculas. Não é possível apenas chamar `lower()` ou passar um argumento, como visto anteriormente, da forma `lower('Hello')`. É necessário anexar o método a alguma string específica, usando um ponto.

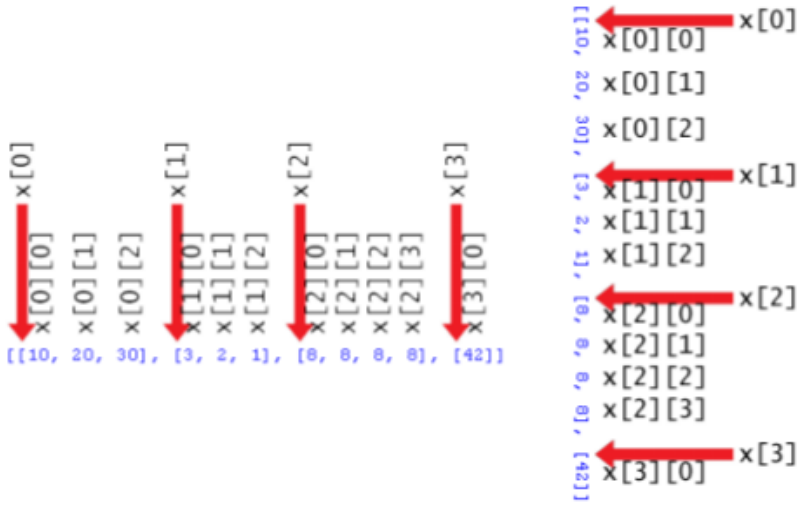

Figura 10: Índices em uma lista que contém outras listas.

Os métodos de string `lower()` e `upper()`

O método `lower()`, conforme mencionado anteriormente, retorna uma cópia de uma string em letras minúsculas. Do contrário, o método `upper()`, retorna uma cópia de uma string em letras maiúsculas. Observe o exemplo:

```
>>> 'Hello world'.lower()
'hello world!'
>>> 'Hello world'.upper()
'HELLO WORLD!'
```

Devido aos métodos retornarem uma string, eles podem ser vistos como uma string e utilizados como tal. Desta forma, podemos fazer o seguinte:

```
>>> 'Hello world'.upper().lower()
'hello world!'
>>> 'Hello world'.lower().upper()
'HELLO WORLD!'
```

Se uma string está armazenada em uma variável, pode-se tratar da mesma forma.

```
>>> fizz = 'Hello world'
>>> fizz.upper()
'HELLO WORLD'
```

6.5.8 Os métodos de lista `reverse()` e `append()`

O tipo de dado de lista também possui métodos. O método `reverse()` inverte a ordem dos itens na lista.

```
>>> spam = [1, 2, 3, 4, 5, 6, 'meow', 'woof']
>>> spam.reverse()
>>> spam
['woof', 'meow', 6, 5, 4, 3, 2, 1]
```

O método de lista mais comum é o `append()`. Este método adiciona um valor passado por parâmetro, no fim da lista.

```
>>> eggs = []
>>> eggs.append('hovercraft')
>>> eggs
['hovercraft']
>>> eggs.append('eels')
>>> eggs
['hovercraft', 'eels']
>>> eggs.append(42)
>>> eggs
['hovercraft', 'eels', 42]
```

6.6 A diferença entre métodos e funções

Você pode se perguntar porque o Python possui métodos, já que eles se comportam como funções. Alguns tipos de dados possuem métodos. Métodos são funções associadas a valores de algum determinado tipo de dado. Por exemplo, métodos de string são funções que podem ser utilizadas (chamadas) por qualquer string.

Não é possível chamar métodos de string para outros tipos de dados. Por exemplo, `[1, 2, 'apple'].upper()` resultaria em um erro, pois `[1, 2, 'apple']` é uma lista e não uma string.

Os valores que possuem métodos são chamados de objetos. Objeto é um conceito de programação orientada a objetos. Não é necessário saber realmente, neste momento, o que é programação orientada a objetos. Vamos explorando os conceitos necessários no decorrer do curso. Por enquanto, é necessário entender que os tipos de dados que possuem métodos são chamados de objetos.

6.6.1 O método de lista `split()`

A linha 59 é uma longa linha de código, mas é apenas uma atribuição de variável. Esta linha também utiliza o método `split()`, também para strings.

```
words = 'ant baboon badger bat bear beaver camel cat clam cobra cougar
    coyote crow deer dog donkey duck eagle ferret fox frog goat goose
    hawk lion lizard llama mole monkey moose mouse mule newt otter owl
    panda parrot pigeon python rabbit ram rat raven rhino salmon seal
    shark sheep skunk sloth snake spider stork swan tiger toad trout
    turkey turtle weasel whale wolf wombat zebra'.split()
```

Como você pode perceber, esta linha é apenas uma longa string, cheia de palavras separadas por espaços. No fim da string, chamamos o método `split()`. Este método transforma a string em uma lista, em que cada palavra se torna um item. A divisão ocorre sempre que houver um espaço. O motivo de utilizarmos esta função, ao invés de declarar uma lista, é que é bem mais fácil escrever uma string longa do que digitar cada palavra como um item de uma lista.

Observe um exemplo:

```
>>> 'My very energetic mother just served us nine pies'.split()
['My', 'very', 'energetic', 'mother', 'just', 'served', 'us', 'nine',
'pies']
```

O resultado é uma lista de nove strings, cada uma para cada palavra da string original. No caso do jogo da forca, a lista `words` conterá cada palavra secreta utilizada na forca. Você pode adicionar suas próprias palavras na string, ou remover também. Apenas tenha certeza de que as palavras estejam separadas por espaços.

6.7 Como o código funciona

Iniciando na linha 61, definimos uma nova função chamada `getRandomWord()`, que posui um único parâmetro chamado `wordlist`. Chamamos esta função para escolher uma palavra secreta da lista `words`.

```
61 def getRandomWord(wordList):
62     # Esta funcao retorna uma string aleatoria da lista de strings
63     wordIndex = random.randint(0, len(wordList) - 1)
64     return wordList[wordIndex]
```

A função `getRandomWord()` recebe uma lista de strings como argumento para o parâmetro `wordList`. Na linha 63, armazenamos um índice aleatório, para ser utilizado na lista recebida, na variável `wordIndex`. Fazemos isto através da função `randint()`, com dois argumentos, que caracterizam a faixa de números que a função poderá escolher. O primeiro argumento é o zero e o segundo é `len(wordList) - 1`. O segundo argumento é uma expressão que resulta em um inteiro. `len(wordList)` retornará o tamanho (um inteiro) da lista passada para a função.

Diminuímos uma unidade do tamanho da lista, pois os itens são identificados a partir do zero. Logo, o último item da lista possuirá índice com o valor da quantidade de itens menos 1.

6.7.1 Mostrando a ilustração para o jogador

```
66 def displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord):
67     print(HANGMANPICS[len(missedLetters)])
68     print()
```

Aqui, foi criada uma função que imprimirá a ilustração do jogo de forca, com as letras que o jogador acertou (ou não). Este código define uma função chamada `displayBoard()`, que recebe quatro parâmetros. Aqui está o que cada parâmetro significa:

- `HANGMANPICS`: é a lista de strings de várias linhas que será mostrada como arte ASCII. Passaremos a variável global `HANGMANPICS` como argumento para este parâmetro.
- `missedLetters`: é uma string de palpites (letras) que o jogador errou.
- `correctLetters`: é uma string de palpites (letras) que o jogador acertou.
- `secretWord`: esta string é a palavra secreta que o jogador deve adivinhar.

A primeira chamada da função `print()` mostrará a ilustração do jogo. `HANGMANPICS` é a lista de strings que possuem os passos do corpo do enforcado.

O número de letras em `missedLetters` armazenará quantos palpites incorretos o jogador fez. Para saber esta quantidade, chamamos `len(missedLetters)`. Este número é utilizado como índice da lista `HANGMANPICS`, para que seja impressa a ilustração correta, de acordo com os palpites incorretos.

```
70     print('Missed letters:', end=' ')
71     for letter in missedLetters:
72         print(letter, end=' ')
73     print()
```

A linha 71 apresenta um novo tipo de loop, o loop `for`. A linha 72 é o que compõe o bloco do loop `for`. A função `range()` é frequentemente utilizada com loops `for`. Veremos esta função adiante.

O argumento `end`

Nas linhas 70 e 72 há um segundo argumento fornecido à função `print()`, o `end=' '`. Este argumento é opcional. Passando uma string em branco para o `end` é o mesmo que dizer à função para que uma nova linha não seja inserida ao fim da string. Ao invés disso, este argumento faz com que um espaço em branco seja adicionado a string impressa. É por isso que a linha seguinte a ser impressa (no caso da linha 70, a linha 72) aparece na mesma linha que a anterior.

6.7.2 As funções `range()` e `list()`

A função `range()` é fácil de entender. Ela pode ser chamada com um ou dois argumentos inteiros. Quando chamada com um argumento, `range()` retornará uma série de valores inteiros, de

zero ao argumento menos uma unidade. Esta série pode ser convertida para uma lista com a função `list()`.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Se forem passados dois argumentos à função `range`, a lista de inteiros retornada compreende desde o primeiro argumento até o segundo argumento menos uma unidade. Observe:

```
>>> list(range(10, 20))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

A função `range` é muito útil, pois frequentemente é usada em loops `for` (que são muito mais poderosos que o loop `while`).

6.7.3 Loops `for`

O loop `for` é uma boa ferramenta para percorrer listas de valores. É diferente do loop `while`, que faz repetições enquanto uma condição é verdadeira. Uma sentença `for` começa com a palavra-chave `for`, seguida por uma variável e a palavra-chave `in`, seguida por uma sequência (como uma lista ou string) ou uma série de valores (retornada pela função `range()`). Para finalizar a sentença, dois-pontos. A cada **iteração**, a variável na sentença `for` recebe o valor do próximo item da lista.

Observe o exemplo:

```
>>> for i in range(10):
...     print(i)
...
0
1
2
3
4
5
6
7
8
9
>>>
```

Com loops `for`, não há necessidade de converter o objeto série (`range()`) para uma lista com o comando `list()`. O loop `for` faz isto automaticamente. Ele executa o código em seu bloco uma vez para cada item da lista. Cada vez que o código é executado, a variável do loop recebe o valor do próximo item da lista. Logo, não é necessário que a iteração do loop possua números. Observe:

```
>>> for thing in ['cats', 'pasta', 'programming', 'spam']:
...     print('I really like ' + thing)
...
I really like cats
I really like pasta
I really like programming
I really like spam
>>> for i in 'Hello world!':
...     print(i)
...
H
e
l
l
o
```

```
w
o
r
l
d
!
```

Outro ponto importante sobre loops `for`, é a utilização da palavra-chave `in`. O uso do `in` em um loop `for` é diferente do que foi visto até então. Aqui, o `in` possui a utilidade de separar a variável da lista em que será feita a iteração.

O trecho seguinte da função `displayBoard()` mostra as letras faltantes e cria a string da palavra secreta de todos os palpites errados, com espaços.

```
70      print('Missed letters:', end=' ')
71      for letter in missedLetters:
72          print(letter, end=' ')
73      print()
```

Se `missedLetters` for igual a `aush`, então este loop imprimirá `a u s h`.

Equivalência de um loop `while` para um loop `for`

O loop `for` é bem semelhante a um loop `while`, mas, quando é necessário realizar uma iteração através de uma lista, um loop `for` torna o código mais sucinto. Ainda assim, um loop `while` pode se comportar da mesma forma que um loop `for` com linhas extras de código. Observe:

```
>>> sequence = ['cats', 'pasta', 'programming', 'spam']
>>> index = 0
>>> while (index < len(sequence)):
...     thing = sequence[index]
...     print('I really like ' + thing)
...     index = index + 1
...
I really like cats
I really like pasta
I really like programming
I really like spam
>>>
```

Perceba que é necessário criar mais código para esta alternativa. Visando tornar o código sempre o mais legível e sucinto, o loop `for` será amplamente utilizado nos próximos programas.

6.7.4 Mostrando os espaços para as letras da palavra secreta

```
75      blanks = '_' * len(secretWord)
76
77      for i in range(len(secretWord)):
78          if secretWord[i] in correctLetters:
79              blanks = blanks[:i] + secretWord[i] + blanks[i+1:]
80
81      for letter in blanks:
82          print(letter, end=' ')
83      print()
```

Até então, já sabemos como mostrar ao jogador os palpites errados. Agora, veremos como imprimir linhas em branco para cada letra da palavra secreta. Para tal, utilizaremos o caracter de sublinhado[17] para as letras não adivinhadas. Inicialmente, uma string com um caractere de sublinhado

[17] *Underscore.*

para cada letra da palavra secreta é criada na linha 75. Observe que é utilizado o operador * com o `_`. Este operador pode ser utilizado entre uma string e um inteiro. Por exemplo, a expressão `'hello' * 3` resulta em `'hellohellohello'`. No programa, a variável `blanks` armazenará em sublinhados o número de caracteres da palavra secreta.

Em seguida, usamos um loop `for` para percorrer cada letra da palavra secreta na variável `secretWord` e substituir o sublinhado com a letra respectiva caso esta exista em `correctLetters`. A linha 79 pode parecer confusa, pois ela utiliza colchetes nas variáveis `blanks` e `secretWords`, que são strings, não listas. Assim como a função `len()`, utilizada anteriormente, apenas aceita listas como parâmetros, não strings. Porém, em Python, muitas coisas feitas com listas também podem sem feitas com strings.

Tarefa 6.1
Pesquise sobre *string slices* em Python.

6.7.5 Substituindo sublinhados por letras adivinhadas corretamente

```
77      for i in range(len(secretWord)):
78          if secretWord[i] in correctLetters:
79              blanks = blanks[:i] + secretWord[i] + blanks[i+1:]
```

Vamos ilustrar este bloco através de um exemplo. Suponha que a palavra secreva em `secretWord` seja `'otter'` e que o jogador adivinhou as letras `tr`, armazenado em `correctLetters`. Assim, `len(secretWord)` retornará 5. Desta forma, `range(len(secretWord))` retornará `[0, 1, 2, 3, 4]`. Devido ao valor de `i` tornar-se cada elemento da série a cada iteração, o loop `for`, neste exemplo se torna o seguinte:

```
if secretWord[0] in correctLetters:
    blanks = blanks[:0] + secretWord[0] + blanks[1:]
if secretWord[1] in correctLetters:
    blanks = blanks[:1] + secretWord[1] + blanks[2:]
if secretWord[2] in correctLetters:
    blanks = blanks[:2] + secretWord[2] + blanks[3:]
if secretWord[3] in correctLetters:
    blanks = blanks[:3] + secretWord[3] + blanks[4:]
if secretWord[4] in correctLetters:
    blanks = blanks[:4] + secretWord[4] + blanks[5:]
```

Reescrevendo:

```
if 'o' in 'tr': # Condição é falsa, blanks == '_____'
    blanks = '' + 'o' + '____' # Esta linha não é executada.
if 't' in 'tr': # Condição é verdadeira, blanks == '_____'
    blanks = '_' + 't' + '___' # Esta linha é executada.
if 't' in 'tr': # Condição é verdadeira, blanks == '_t___'
    blanks = '_t' + 't' + '__' # Esta linha é executada.
if 'e' in 'tr': # Condição é falsa, blanks == '_tt__'
    blanks = '_tt' + 'e' + '_' # Esta linha não é executada.
if 'r' in 'tr': # Condição é verdadeira, blanks == '_tt__'
    blanks = '_tt_' + 'r' + '' # Esta linha é executada.
# blanks agora é equivalente a '_tt_r'
```

As próximas linhas de código mostram o valor armazenado na variável `blanks` com espaços entre cada caracter.

```
81      for letter in blanks:
82          print(letter, end=' ')
83      print()
```

6.7.6 Obtendo os palpites do jogador

A função `getGuess()` será chamada a cada vez que o jogador tiver que digitar uma letra. A função retorna a o palpite como uma string. Além disso, `getGuess()` faz com que o jogador digite apenas uma letra, antes de fazer o seu retorno.

```
85 def getGuess(alreadyGuessed):
86
87     while True:
88         print('Guess a letter.')
89         guess = input()
90         guess = guess.lower()
91         if len(guess) != 1:
92             print('Please enter a single letter.')
93         elif guess in alreadyGuessed:
94             print('You have already guessed that letter. Choose again.')
95         elif guess not in 'abcdefghijklmnopqrstuvwxyz':
96             print('Please enter a LETTER.')
97         else:
98             return guess
```

Esta função possui um parâmetro chamado `alreadyGuessed` que contém as letras que o jogador já adivinhou. Ela irá pedir ao jogador para que o palpite seja apenas uma letra. Este palpite será o retorno da função.

Usamos o loop `while` pois o palpite do jogador deverá ser uma e somente uma letra que ele não tenha tentado adivinhar anteriormente. Observe que a condição para o loop `while` é apenas o valor booleano `True`. Isto significa que a execução sairá do loop somente com um `break`, que para imediatamente a função, ou `return`, que sai da função assim que obtém um resultado ou cumpre alguma condição. Este tipo de loop é chamado de **infinito**.

O código dentro desta função pede um palpite para o jogador, que é armazenado na variável `guess`. Se o jogador entrar com uma letra maiúscula, ela será convertida para minúscula com o comando da linha 90.

6.8 Sentenças `elif` (`else if`)

Observe o seguinte código:

```
if catName == 'Fuzzball':
    print('Your cat is fuzzy.')
else:
    print('Your cat is not very fuzzy at all.')
```

É um código bem simples: se a variável `catName` for igual à string `'Fuzzball'` imprimirá algo, do contrário, imprimirá outra coisa.

Entretanto, apenas duas escolhas podem não ser o suficiente para a prática da programação. Desta forma, podemos adaptar o `if-else` anterior para algo do tipo:

```
if catName == 'Fuzzball':
    print('Your cat is fuzzy.')
else:
    if catName == 'Spots'
        print('Your cat is spotted.')
    else:
        if catName == 'FattyKitty'
            print('Your cat is fat.')
        else:
            if catName == 'Puff'
                print('Your cat is puffy.')
            else:
```

```
                print('Your cat is neither fuzzy nor spotted nor
                    fat nor puffy.')
```

Contudo, digitar esta porção de espaços não é nada eficiente e mais suscetível a erros. Para várias opções, o Python fornece o `elif`. Desta forma, o código anterior toma a seguinte forma:

```
if catName == 'Fuzzball':
    print('Your cat is fuzzy.')
elif catName == 'Spots'
    print('Your cat is spotted.')
elif catName == 'FattyKitty'
    print('Your cat is fat.')
elif catName == 'Puff'
    print('Your cat is puffy.')
else:
    print('Your cat is neither fuzzy nor spotted nor fat nor puffy.')
```

Se a condição do primeiro `if` for falsa, então o programa verificará a próxima condição, do primeiro bloco `elif`. Se esta for falsa também, o programa verificará a próxima condição e assim por diante, até chegar no último `else`, que é executado quando não há mais condições a serem verificadas.

Se uma condição for verdadeira, no bloco como um todo, o sub-bloco, cuja condição for verdadeira, é executado e todos os outros são ignorados. Ou seja, apenas um bloco é executado em uma sequência de `if-elif-else`. A última saída `else` pode ser omitida, caso não haja situação cabível a uma alternativa geral.

6.8.1 Ter certeza que o jogador deu um palpite válido

```
91          if len(guess) != 1:
92              print('Please enter a single letter.')
93          elif guess in alreadyGuessed:
94              print('You have already guessed that letter. Choose again.')
95          elif guess not in 'abcdefghijklmnopqrstuvwxyz':
96              print('Please enter a LETTER.')
97          else:
98              return guess
```

A variável `guess` possui o texto que o jogador deu como palpite. Precisamos ter certeza de que este palpite seja uma única letra, do contrário, pedimos novamente para que ele digite uma entrada válida. Desta forma, temos o bloco da função `getGuess()` a partir da linha 91.

Este bloco verifica:

1. Se a entrada possui apenas um caracter, através da função `len(guess)`
2. Se a entrada é repetida, através da verificação da string `guess` em `alreadyGuessed`
3. Se a entrada é uma letra e não qualquer outro caracter

Todas as verificações resultam em uma mensagem de erro, caso sejam verdadeiras. Na última alternativa, o palpite do jogador é retornado. Isto quer dizer nenhuma das condições anteriores é verdadeira, logo, o jogador deu uma entrada válida para o programa, no caso, uma letra. Lembre-se que apenas um destes blocos é executado.

6.8.2 Perguntando se o jogador deseja jogar novamente

```
100 def playAgain():
101
102     print('Do you want to play again? (yes or no)')
103     return input().lower().startswith('y')
```

A função `playAgain()` apenas imprime uma frase perguntando se o jogador deseja permanecer no jogo e retorna o resultado baseado no que o jogador digitar.

Embora a expressão de retorno pareça complicada, ela é bem simples. Ela não possui operadores, mas há uma chamada de função e duas chamadas de métodos. A função é a `input()` e os métodos são `lower()` e `startswith()`. A função retorna o texto que o jogador digitar. A partir desta string, o método `lower()` a transformará em letras minúsculas e o método `startswith()` verificará se a string inicia com a letra `'y'`. Desta forma, a expressão, como um todo, retorna um valor booleano[18].

6.9 Revisão das funções criadas para este jogo

1. `getRandomWord(wordList)`: através de uma lista de strings, sorteará uma e retornará como palavra secreta do jogo
2. `displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)`: mostra o estado atual do jogo, incluindo ilustração, o quanto a palavra secreta foi adivinhada e os palpites equivocados. Esta função precisa de quatro argumentos: `HANGMANPICS` é a lista de strings que contém a ilustração da forca; `correctLetters` e `missedLetters` são as strings das letras que o jogador adivinhou e as que ele não adivinhou, respectivamente; e, `secretWord` é a palavra secreta que o jogador deverá adivinhar.
3. `getGuess(alreadyGuessed)`: através de uma string de letras que o jogador já tentou adivinhar, a função continua pedindo ao jogador uma letra que ele ainda não tenha dado como palpite. Por fim, o palpite válido é retornado.
4. `playAgain()`: é a função que pergunta ao jogador que ele gostaria de jogar outra rodada da forca. Retorna `True` se o jogador quiser e `False`, do contrário.

6.10 Código principal do jogo

O código que precisamos escrever foi ilustrado na figura 9. Boa parte já foi feita, mas ainda é necessário colocar tudo isso em ordem e saber como chamar as funções. Assim, tudo começa a ser executado, na verdade, na linha 106.

```
106 print('H A N G M A N')
107 missedLetters = ''
108 correctLetters = ''
109 secretWord = getRandomWord(words)
110 gameIsDone = False
```

Neste primeiro bloco, apenas é impresso o nome do jogo e atribuído uma string vazia para as variáveis `missedLetters` e `correctLetters`. Em seguida, uma palavra escolhida é atribuída à variável `secretWord`, através da lista definida anteriormente, atribuída à variável `words`. Finalizando, é atribuído o valor booleano `False` para a variável `gameIsDone`. Todo este processo é feito antes de o jogador dar o seu primeiro palpite.

Para alterar o valor desta última variável, o jogo deverá ser terminado e o programa deverá perguntar ao jogador se ele gostaria de jogar novamente.

6.10.1 Mostrando a ilustração ao jogador

O loop `while` possui a condição sempre verdadeira, assim, tudo o que está no bloco será executado até que seja encontrado um `break`. Isto é feito quando o jogo acaba (se o jogador perder ou vencer o jogo).

```
112 while True:
113     displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)
```

A linha 113 chama a função `displayBoard()`, passando os estados da forca em forma de string e as variáveis criadas anteriormente. De acordo com quantas letras o jogador adivinhou ou não, esta função mostra o estado atual da forca ao jogador.

[18] Ainda há o método `endswith()`, que verifica o fim da string.

6.10.2 Permitindo o jogador a fazer um palpite e verificando se o palpite está na palavra secreta

```
116     guess = getGuess(missedLetters + correctLetters)
117
118     if guess in secretWord:
119         correctLetters = correctLetters + guess
```

A linha 116 chama a função `getGuess()` passando os palpites do jogador como argumento. Observe que tanto os acertos quanto os erros são enviados à função. Isto é feito pois a função verifica tanto os palpites corretos quanto os errados, e, dependendo do próximo palpite do jogador, o programa se comporta de uma forma diferente.

Em seguida, é feita a verificação de se o palpite (válido) do jogador está presente na palavra sorteada. Se esta condição for verdadeira, então o bloco `if` será executado e a variável que carrega os acertos do jogador é atualizada.

6.10.3 Verificar se o jogador venceu ou perdeu o jogo

O código entre as linhas 121 e 137 faz a verificação se o jogador ganhou ou perdeu o jogo. Primeiro, discutiremos o bloco que verifica se o jogador venceu, em seguida, se o jogador perdeu.

Como saber se o jogador adivinhou todas as letras da palavra secreta? `correctLetters` possui cada letra que o jogador acertou, enquanto `secretWord` possui a palavra a ser adivinhada. Não podemos fazer a verificação `correctLetters == secretWord` porque as letras não são adivinhadas na ordem, e podem haver repetições de uma mesma letra. Desta forma, cada letra em `secretWord` é comparada às letras de `correctLetters` para verificar se o jogador venceu o jogo. Este processo é executado entre as linhas 122 e 126.

Observe que a verificação de `secretWord` em `correctLetters` é diferente da verificação de `correctLetters` em `secretWord`. O conjunto de palpites armazenado em `correctLetters` pode ser um subconjunto de `secretWord`, o que não quer dizer que o jogador adivinhou a palavra corretamente.

Então, na prática, como é feita a verificação? Inicialmente, assumimos que todas as letras foram adivinhadas, em `foundAllLetters = True`. O estado desta variável é alterado no loop `for` seguinte, quando executar bloco `if` dentro do loop. No caso de `foundAllLetters` não ter sido alterada, o jogador venceu o jogo e o bloco `if` seguinte é executado.

Após esta verificação e o jogador ter ganhado o jogo, o programa deverá informar isto ao jogador. Desta forma, uma variável `gameIsDone` é designada como `True`. Esta variável é utilizada para o programa saber se o jogador deverá continuar seus palpites ou encerrar.

No caso de o jogador ter dado palpites incorretos, a execução cairá no bloco `else`. Para este bloco ser executado, a condição do bloco `if` anterior `if guess in secretWord:`, na linha 118. Devido ao palpite do jogador ter sido errado, ele será armazenado em `missedLetters`. Em seguida, é verificado se o jogador cometeu o máximo de erros possível. Se a condição for verdadeira, então é mostrado a última ilustração para a forca com uma mensagem de que o jogador perdeu o jogo. Novamente, a variável que verifica se o jogo chegou ao fim `gameIsDone` recebe o valor `True`.

6.10.4 Iniciando uma nova rodada

Se o jogador ganhou ou perdeu o jogo, ele poderá escolher entre continuar o jogo ou não. Para isto, temos o último bloco `if`. Este bloco será sempre executado quando um jogo terminar, pois ele exige que a variável `gameIsDone` tenha o valor `True`, que é feito em ambos os blocos que verificam se o jogador ganhou ou perdeu o jogo.

Logo, a execução cairá no bloco `if playAgain():`, que perguntará se o jogador deseja jogar novamente. Em caso afirmativo, as variáveis iniciais são resetadas e o ciclo recomeça, pois ainda estamos no bloco `while` principal. No caso de o jogador não quiser uma nova rodada, ocorre um `break` e o programa termina.

6.11 Resumo

Este foi um longo capítulo e, finalmente, terminamos. Muitos conceitos foram introduzidos e este jogo foi o mais avançado até agora. Como os programas se tornarão mais complexos, uma boa forma de se organizar é através de fluxogramas. Cinco minutos desenhando um fluxograma podem economizar horas de produção de código.

Entre os conceitos abordados tivemos:

1. Métodos: funções associadas com valores. O valor de retorno dos métodos dependem dos valores aos quais o método está associado.
2. Loops `for` iteram através de cada valor da lista. A função `range()` é frequetemente utilizada com estes loops, pois é uma forma fácil para criar listas de números sequenciais.
3. Sentenças `elif` são executadas se a condição do bloco for verdadeira e as condições anteriores forem falsas.

Tarefa 6.2
Pesquise sobre o tipo *dictionary* e a diferença do tipo *list* (lista). Desenvolva como o *dictionary* pode ser utilizado neste jogo de forca.

Tarefa 6.3
Pesquise sobre múltiplas atribuições de variáveis no Python.

6.12 Exercícios complementares

1. Faça um programa que leia uma lista de 10 números reais e mostre-os na ordem inversa.
2. Faça um programa que simule um lançamento de dados. Lance o dado 100 vezes e armazene os resultados em uma lista. Depois, mostre quantas vezes cada valor foi conseguido. Dica: use uma lista de contadores (1-6) e uma função para gerar numeros aleatórios, simulando os lançamentos dos dados.
3. Faça um programa que leia duas listas com 10 elementos cada. Gere uma terceira lista de 20 elementos, cujos valores deverão ser compostos pelos elementos intercalados das duas outras listas.
4. **Nome na vertical**. Faça um programa que solicite o nome do usuário e imprima-o na vertical.
5. Faça um programa que leia uma sequência de caracteres, mostre-a e diga se é um palíndromo[19] ou não.
6. **Jogo da palavra embaralhada**. Desenvolva um jogo em que o usuário tenha que adivinhar uma palavra que será mostrada com as letras embaralhadas. O programa terá uma lista de palavras e escolherá uma aleatoriamente. O jogador terá seis tentativas para adivinhar a palavra. Ao final a palavra deve ser mostrada na tela, informando se o usuário ganhou ou perdeu o jogo.

[19] Um palíndromo é uma sequência de caracteres cuja leitura é idêntica se feita da direita para esquerda ou vice-versa

7 Jogo da velha

Tópicos abordados neste capítulo:

- Introdução à inteligência artifical
- Referências de lista
- O valor `None`

Aprenderemos agora a criar um jogo da velha, em que o jogador enfrentará uma inteligência artificial simples. Um programa com Inteligência Artificial (**IA**[20]) é um programa de computador que pode responder "com inteligência" para os movimentos do jogador. Este jogo não introduz nenhum conceito muito difícil, apenas mais linhas de código a serem analisadas.

O jogo da velha é um jogo muito simples, geralmente jogado com caneta e papel, por duas pessoas. Uma pessoa representa suas jogadas por um X e outra por um círculo. Aqui, ilustraremos estes símbolos com as letras X e O. Se o jogador fizer uma combinação de três símbolos em uma linha ou diagonal, ele vence.

O princípio deste jogo aqui é o mesmo. A "inteligência artificial" será utilizada para o computador confrontar o jogador. Ou seja, não haverá duas pessoas, como de costume, neste jogo. Será o jogador contra o computador.

Este capítulo não introduz muitos conceitos novos, mas aprimora a habilidade para desenvolver jogos. Afinal, o principal de um jogo não é apenas o uso de recursos gráficos e sonoros, mas uma boa lógica, funcionando adequadamente.

7.1 Modelo de saída do jogo da velha

```
Welcome to Tic Tac Toe!
Do you want to be X or O?
O
The computer will go first.

   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   | X
   |   |
What is your next move? (1-9)
4
   |   |
   |   | X
   |   |
-----------
   |   |
 O |   |
   |   |
-----------
   |   |
   |   | X
   |   |
```

[20] **AI**, em inglês, de *artificial intelligence*.

```
What is your next move? (1-9)
6
   |   |
   |   | X
   |   |
-----------
   |   |
 O | X | O
   |   |
-----------
   |   |
   |   | X
   |   |
What is your next move? (1-9)
1
   |   |
 X |   | X
   |   |
-----------
   |   |
 O | X | O
   |   |
-----------
   |   |
 O |   | X
   |   |
The computer has beaten you! You lose.
Do you want to play again? (yes or no)
no
```

7.2 Código-fonte do jogo da velha

Código 5: Jogo da velha.

```
# Jogo da velha

import random

def drawBoard(board):
    # Esta funcao imprime o tabuleiro do jogo

    # "board" eh uma lista de 12 strings representando o tabuleiro (ignorando
        o indice 0)
    print('   |   |')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
    print('   |   |')

def inputPlayerLetter():
    # Deixa o jogador escolher com qual letra ele gostaria de ser
        representado  no jogo
    # Retorna uma lista com a letra que o jogador escolheu como o primeiro
        item e a do computador como o segundo
    letter = ''
    while not (letter == 'X' or letter == 'O'):
        print('Do you want to be X or O?')
        letter = input().upper()

    # o primeiro elemento na tupla eh a letra do jogador, a segunda eh a do
        computador
    if letter == 'X':
        return ['X', 'O']
    else:
        return ['O', 'X']

def whoGoesFirst():
    # Aleatoriamente escolhe quem o jogador que inicia o jogo
    if random.randint(0, 1) == 0:
        return 'computer'
    else:
        return 'player'

def playAgain():
    # Esta funcao retorna True se o jogador quiser jogar novamente.
    print('Do you want to play again? (yes or no)')
    return input().lower().startswith('y')

def makeMove(board, letter, move):
    board[move] = letter

def isWinner(bo, le):
    # Esta funcao retorna True se o jogador vencer o jogo
    # Usamos bo, ao inves de board, e le, ao inves de letter, para que nao
        precisemos digitar tanto
    return ((bo[7] == le and bo[8] == le and bo[9] == le) or
```

```
54      (bo[4] == le and bo[5] == le and bo[6] == le) or
55      (bo[1] == le and bo[2] == le and bo[3] == le) or
56      (bo[7] == le and bo[4] == le and bo[1] == le) or
57      (bo[8] == le and bo[5] == le and bo[2] == le) or
58      (bo[9] == le and bo[6] == le and bo[3] == le) or
59      (bo[7] == le and bo[5] == le and bo[3] == le) or
60      (bo[9] == le and bo[5] == le and bo[1] == le))
61
62  def getBoardCopy(board):
63      # Faz uma copia da lista do tabuleiro e retorna
64      dupeBoard = []
65
66      for i in board:
67          dupeBoard.append(i)
68
69      return dupeBoard
70
71  def isSpaceFree(board, move):
72      # Retorna True se a jogada esta livre no tabuleiro
73      return board[move] == ' '
74
75  def getPlayerMove(board):
76      # Permite ao jogador digitar seu movimento
77      move = ' '
78      while move not in '1 2 3 4 5 6 7 8 9'.split() or not isSpaceFree(board,
            int(move)):
79          print('What is your next move? (1-9)')
80          move = input()
81      return int(move)
82
83  def chooseRandomMoveFromList(board, movesList):
84      # Retorna um movimento valido da lista passada no tabuleiro
85      possibleMoves = []
86      for i in movesList:
87          if isSpaceFree(board, i):
88              possibleMoves.append(i)
89
90      if len(possibleMoves) != 0:
91          return random.choice(possibleMoves)
92      else:
93          return None
94
95  def getComputerMove(board, computerLetter):
96      # Dado um tabuleiro e o simbolo do jogador, a funcao determina onde jogar
             e retorna o movimento
97      if computerLetter == 'X':
98          playerLetter = 'O'
99      else:
100         playerLetter = 'X'
101
102     # Aqui esta o algoritmo para a 'inteligencia artificial' do jogo da velha
103     # Primeiro, verificamos se eh possivel vencer na proxima jogada
104     for i in range(1, 10):
105         copy = getBoardCopy(board)
106         if isSpaceFree(copy, i):
107             makeMove(copy, computerLetter, i)
108             if isWinner(copy, computerLetter):
109                 return i
110
111     # Verifica se o jogador pode vencer na proxima jogada e, entao, o
            bloqueia
112     for i in range(1, 10):
```

```
113             copy = getBoardCopy(board)
114             if isSpaceFree(copy, i):
115                 makeMove(copy, playerLetter, i)
116                 if isWinner(copy, playerLetter):
117                     return i
118 
119         # Tenta ocupar algum dos cantos, se eles estiverem livres
120         move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
121         if move != None:
122             return move
123 
124         # Tenta ocupar o centro, se estiver livre
125         if isSpaceFree(board, 5):
126             return 5
127 
128         # Ocupa os lados
129         return chooseRandomMoveFromList(board, [2, 4, 6, 8])
130 
131 def isBoardFull(board):
132         # Retorna True se todos os espacos do tabuleiro estiverem ocupados
133         for i in range(1, 10):
134             if isSpaceFree(board, i):
135                 return False
136         return True
137 
138 
139 print('Welcome to Tic Tac Toe!')
140 
141 while True:
142         # Reinicia o tabuleiro
143         theBoard = [' '] * 10
144         playerLetter, computerLetter = inputPlayerLetter()
145         turn = whoGoesFirst()
146         print('The ' + turn + ' will go first.')
147         gameIsPlaying = True
148 
149         while gameIsPlaying:
150             if turn == 'player':
151                 # Vez do jogador
152                 drawBoard(theBoard)
153                 move = getPlayerMove(theBoard)
154                 makeMove(theBoard, playerLetter, move)
155 
156                 if isWinner(theBoard, playerLetter):
157                     drawBoard(theBoard)
158                     print('Hooray! You have won the game!')
159                     gameIsPlaying = False
160                 else:
161                     if isBoardFull(theBoard):
162                         drawBoard(theBoard)
163                         print('The game is a tie!')
164                         break
165                     else:
166                         turn = 'computer'
167 
168             else:
169                 # Vez do computador
170                 move = getComputerMove(theBoard, computerLetter)
171                 makeMove(theBoard, computerLetter, move)
172 
173                 if isWinner(theBoard, computerLetter):
174                     drawBoard(theBoard)
```

```
175                 print('The computer has beaten you! You lose.')
176                 gameIsPlaying = False
177             else:
178                 if isBoardFull(theBoard):
179                     drawBoard(theBoard)
180                     print('The game is a tie!')
181                     break
182                 else:
183                     turn = 'player'
184
185     if not playAgain():
186         break
```

7.3 Projetando o programa

O jogo da velha é um jogo muito fácil e rápido, quando jogado no papel entre duas pessoas. No computador, este processo é um pouco mais complicado, pois devemos "mandá-lo" executar passo-a-passo o que quisermos que aconteça.

Desta forma, projetaremos este jogo utilizando um fluxograma. Observe a figura 11.

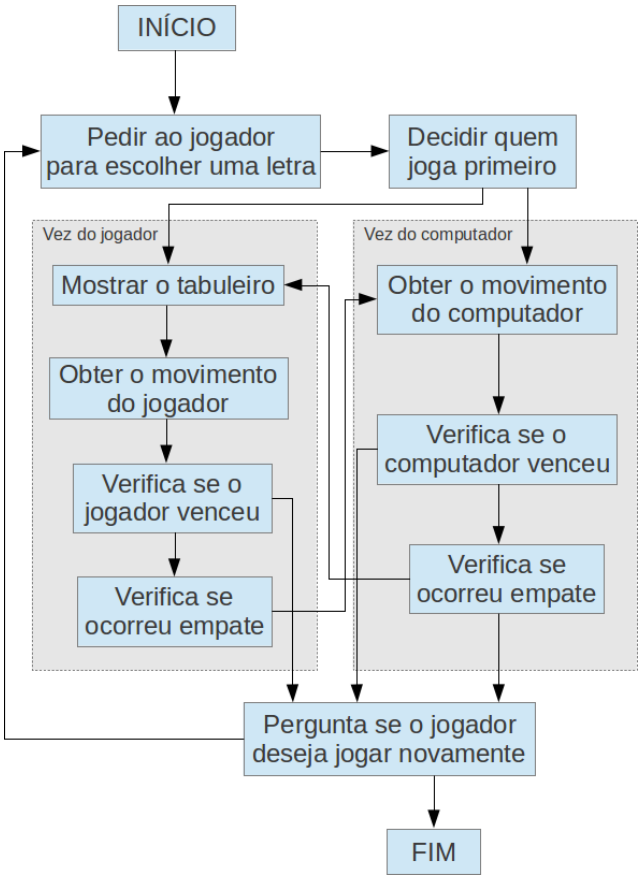

Figura 11: Fluxograma para o jogo da velha.

7.4 Representando o tabuleiro

Primeiro, devemos descobrir como representar o tabuleiro como uma variável. No papel, cruzamos duas linhas horizontais com duas verticais e preenchemos os espaços em branco, com cada símbolo representando cada jogador.

No programa, vamos representá-lo como uma lista de strings. Cada string representará uma das nove posições do tabuleiro. Iremos numerá-las da mesma forma que um teclado numérico, para que seja fácil lembrarmos cada posição. Veja a figura 12

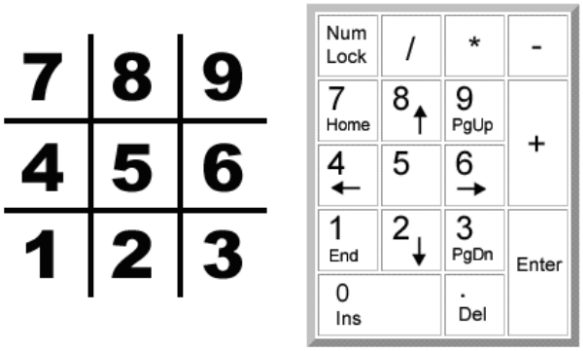

Figura 12: Representação do tabuleiro para o jogo da velha.

Os símbolos que representarão os jogadores serão as strings `'X'` e `'O'` e uma string de espaço `' '` representará as casas em que ainda não foi feita uma jogada.

7.5 Raciocínio do jogo

Para o computador responder ao jogo, devemos identificar os locais em que as jogadas serão feitas. Para tal, identificamos o tabuleiro de acordo com o esquema da figura 13.

Figura 13: Representação do tabuleiro do jogo da velha para o computador.

O raciocínio do jogo (IA) segue um algoritmo simples. Um **algoritmo** é uma série de instruções que executam algo. Logo, tudo o que fizemos até agora, baseou-se em algoritmos. No caso deste jogo da velha, o algoritmo determinará os passos que definem a melhor jogada.

Este algoritmo possui os seguintes passos:

1. Primeiro, verificar se o próximo movimento do computador pode ganhar o jogo. Em caso afirmativo, executar este movimento. Do contrário, seguir próximo passo.
2. Verifica se há algum movimento do jogador que possa fazê-lo ganhar o jogo. Se houver, o computador bloqueia o jogador. Do contrário, seguir próximo passo.
3. Verifica se há espaços livres nos cantos (posições 1, 3, 7, ou 9) e ocupa um deles. Caso não haja, seguir próximo passo.
4. Verifica se o centro está livre. Se estiver, a jogada vai para esta posição. Caso contrário, seguir próximo passo.
5. A jogada irá para qualquer outra posição dos lados (2, 4, 6 ou 8). Não há mais passos, pois, ao chegar nesta alternativa, o computador já verificou se poderia ocupar qualquer outra posição disponível no tabuleiro.

Todo este procedimento é o que ocorre na caixa do fluxograma da figura 11 que indica "obter o movimento do computador". Este passo pode ser expandido através dos itens 1 a 5 anteriores. Observe a figura 14.

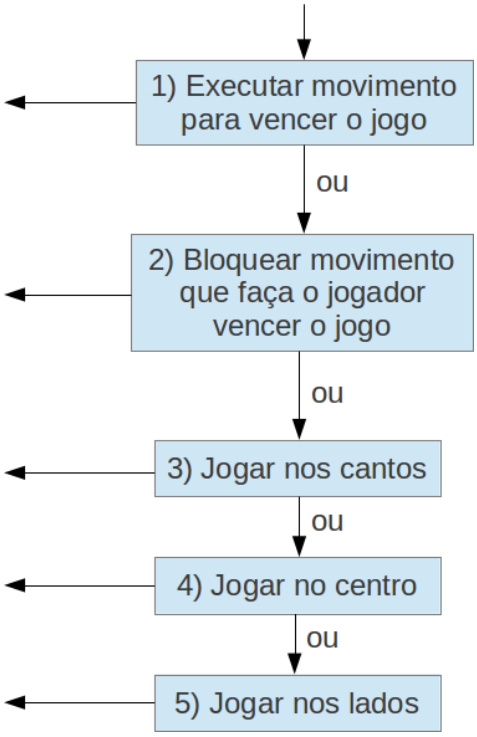

Figura 14: Expansão do passo "Obter o movimento do computador" no fluxograma.

Este algoritmo é implementado em `getComputerMove()` e nas funções que esta função chama.

7.6 Como o programa funciona

A partir de agora, as explicações dos códigos tornar-se-ão mais sucintas. Até este ponto, você já deve ter desenvolvido a habilidade de observar e compreender o que está acontecendo no programa, através da análise do código. Não repetiremos mais tantas linhas de código ou mostraremos exemplos extensos, pois supomos que a sua capacidade de investigação também tenha se desenvolvido. Desta forma, a explicação se extenderá através dos pontos principais do programa.

7.6.1 Imprimindo o tabuleiro na tela

A função `drawBoard()` imprime o tabuleiro na tela, recebendo o parâmetro `board`. Lembre-se que o nosso tabuleiro (`board`) é representado por uma lista de dez strings e que ignoramos a string com o índice 0, para que utilizemos os índices das strings de acordo com a posição do tabuleiro, ilustrada na figura 12. Outras funções também funcionarão através deste parâmetro de dez strings.

Tenha certeza de que os espaços do tabuleiro estejam dispostos corretamente, ou a ilustração não sairá adequada. Programas de computador apenas fazem exatamente o que você os manda fazer, mesmo que esteja errado. Por isso, preste atenção em cada passo do programa.

7.6.2 Permitindo o jogador a escolher o símbolo desejado

A função `inputPlayerLetter()` é bem simples, que apenas pergunta se o jogador deseja ser representado por X ou O e mantém a pergunta até que o jogador escolha uma alternativa válida (através do loop `while`). Observe que a linha 27 transforma a entrada do jogador em letra maiúscula, com o método de string `upper()`.

Em seguida, a função retorna uma lista de dois itens. O primeiro item (índice 0) será o símbolo escolhido pelo jogador; o segundo, o símbolo do computador. O bloco `if-else` entre as linhas 30 e 33 determina qual lista retornar.

7.6.3 Decidindo quem inicia o jogo

A função `whoGoesFirst()` funciona como um jogo de cara ou coroa virtual. Através da função `randint()`, sorteamos um número que será ou 0 ou 1. Se der 0, o computador inicia o jogo. Se der 1, o jogador inicia. Observe que `whoGoesFirst()` retorna uma string.

7.6.4 Perguntando se o jogador deseja jogar novamente

Esta opção é a mesma do jogo da forca e foi apresentada no 6. Desta forma, reaproveitamos o código (no caso, a função) do código 4 e já podemos perceber como funções podem ser realmente úteis.

7.6.5 Mostrando a jogada no tabuleiro

A função `makeMove()` é muito simples e é desenvolvida em apenas uma linha. Os parâmetros que ela recebe são: o tabuleiro (com as dez strings que o representam), `board`; uma das letras que representam a jogada do usuário, `letter`; e um inteiro de 1 a 9, `move`.

Mas... Espere! Por utilizarmos uma função para mudar o valor de um item de `board`, ela não esquecerá o valor alterado logo que sair da função? A resposta é não. Listas são tratadas de uma forma especial quando passadas para funções. Isto é porque não passamos a lista em si para uma função, mas uma referência dela.

7.7 Referências de lista

Entre com as seguintes linhas no terminal interativo:

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

Estas linhas fazem sentido, baseado no que aprendemos até agora. Para refrescar a memória: atribuímos 42 à variável `spam` e então fazemos uma cópia deste valor para a variável `cheese`. Posteriormente, alteramos o valor da variável `spam` para 100, o que não afeta o valor em `cheese`. Isto ocorre porque `spam` e `cheese` são variáveis diferentes, que armazenam valores diferentes.

Contudo, quando se diz respeito à listas, o processo é um pouco diferente. Quando uma lista é atribuída a uma variável com o sinal de igual =, na verdade, o que está sendo atribuído à nova variável é uma referência à lista. Uma **referência** é um valor que aponta a algum tipo de dado, logo, uma **referência de lista** é um valor que aponta para uma lista.

Observe:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = spam
>>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

A linha `cheese = spam` copia uma *referência da lista* de `spam` para `cheese` ao invés de copiar a lista inteira. Isto ocorre porque o valor armazenado em `spam` não é uma lista, mas a referência para uma lista. Isto significa que os valores atribuídos à `spam` e `cheese` referem-se à mesma lista, logo, os valores da lista não foram copiados. Assim, se o valor da lista referenciada por `cheese` for modificado, o valor da lista referenciada por `spam` também muda, pois há apenas uma lista.

Em `makeMove()`, quando passamos `board` como parâmetro, a variável local recebe uma cópia da referência, não a lista como um todo. `letter` e `move` são variáveis que recebem cópias dos valores

passados como argumento (uma string e um inteiro). Desta forma, é possível alterar os valores de `letter` e `move` sem alterar as variáveis originais.

Entretanto, ao alterarmos o valor de alguma das posições da lista em `board`, o valor original será alterado. Ao fim da função, a referência que está em `board` será esquecida, mas a alteração na lista permanecerá.

7.8 Verificar se o jogador venceu

A função `isWinner()` define as jogadas que fazem um jogador vencedor, ou seja, os trios de símbolos iguais, de acordo com as regras do jogo da velha. Para isso, é fornecido o tabuleiro no estado atual e o símbolo que representa um jogador. Esta função é, na verdade, um longo bloco `if`.

Há oito formas de vencer o jogo da velha, entre diagonais, verticais e horizontais. Observe que cada linha da função possui uma destas combinações, com cada item interligado com o operador `and`, e cada uma das oito combinações está interligada com o operador `or`. Isto quer dizer que apenas uma das oito combinações deve ser verdadeira para que o jogador representado pela variável `le` seja o vencedor.

7.9 Duplicando o tabuleiro

A função `getBoardCopy()` faz uma cópia das strings que representam o tabuleiro do jogo da velha. Isto é feito pois há momentos em que o algoritmo de inteligência artificial fará modificações temporárias e isto não deverá afetar o tabuleiro original.

A linha 64 cria uma lista vazia e a referencia em `dupeboard`. O loop `for` seguinte percorrerá o tabuleiro, passado por parâmetro, adicionando uma cópia do tabuleiro original para `dupeboard`, que é retornado ao fim da função.

7.10 Verificando se um espaço no tabuleiro está livre

A função `isSpaceFree()` retorna se o espaço requisitado está livre ou não. Lembre-se que os espaços livres do tabuleiro são marcados com uma string que possui apenas um espaço.

7.11 Permitindo que o jogador entre com a sua jogada

A função `getPlayerMove()` pede ao jogador para entrar com o número do respectivo espaço que ele deseja jogar. A função assegura que a entrada seja válida (um inteiro de 1 a 9) e também verifica se o espaço escolhido já está ocupado.

7.12 Escolhendo um movimento da lista de movimentos

A função `chooseRandomMoveFromList()` é utilizada para a implementação da inteligência artificial. O primeiro parâmetro `board` é a lista de 10 strings, que representam o tabuleiro do jogo. o segundo parâmetro `movesList` é a lista de inteiros que representam os movimentos possíveis.

Esta função escolherá um dos movimentos da lista `possibleMoves`, que carrega todos os movimentos possíveis do tabuleiro, obtidos através da verificação pela função `isSpaceFree()`. A cada verificação de um movimento que for ume espaço vazio, este é adicionado à lista `possiblemoves` com o método `append()`.

Considerando a possibilidade de esta lista estar vazia, ou seja, se não houver mais espaços disponíveis e o tamanho da lista (o número de elementos) for igual a zero, o bloco `if-else`, a partir da linha 91, retornará `None`.

7.12.1 O valor `None`

`None` é um valor especial que pode ser atribuído a uma variável. Ele representa a falta de um valor, e é único do tipo *NoneType* (assim como os valores booleanos são apenas dois, o *NoneType* só

possui um valor). Este valor pode ser útil quando não há uma atribuição válida para uma variável. Por exemplo, se, em um jogo de perguntas e respostas a pessoa tem a opção de não responder a alguma questão, a variável que carrega a resposta pode receber o valor `None`.

7.13 Criando a inteligência artificial

A função `getcomputerMove()` é onde o algoritmo será implementado. Os parâmetros são o tabuleiro `board` e qual letra representará o computador. Esta função retorna o inteiro que representa o espaço que o computador irá jogar.

Lembre-se de como o algoritmo funciona, ele foi descrito na seção 7.5.

7.13.1 O computador verifica se pode vencer em um movimento

O loop `for` iniciado na linha 105, verifica todos os espaços, para saber se o computador pode vencer na próxima jogada. A primeira linha do loop faz uma cópia do tabuleiro e então verifica se alguma das jogadas possíveis pode vencer o jogo, se puder, é retornada esta jogada para o tabuleiro original.

Se não há jogadas possíveis para uma vitória imediata, o loop termina e o próximo passo é executado.

7.13.2 O computador verifica se o jogador pode vencer em uma jogada

A partir da linha 113, há um loop similar ao anterior, mas, desta vez, o computador verifica se o jogador poderá vencer na próxima jogada. Se for, o computador bloqueia a jogada. Se não houver como, o loop termina e o computador decidirá a próxima jogada de outra forma.

7.13.3 Verificando os cantos, o centro e os lados (nesta ordem)

Na linha 121, a chamada da função `chooseRandomMoveFromList()` com a lista `[1, 3, 7, 9]`, vai retornar um inteiro correspondente a um dos espaços dos cantos. Se todos os espaços estiverem tomados, o retorno será `None` e a execução continua na linha 125, que verifica se o centro está ocupado. Caso nenhuma das opções anteriores seja satisfeita, o computador escolherá um dos lados do tabuleiro, também chamando a função `chooseRandomMoveFromList()`, mas com a lista que indica a posição dos lados como parâmetro.

Assim, a função que implementa a inteligência artificial do jogo é encerrada. A cada jogada ela será executada novamente.

7.14 Verificando se o tabuleiro está cheio

A última função escrita é `isBoardFull()` que retorna `True` se a lista de strings que possui as jogadas `'X'` e `'O'` estiver cheia (excetuando-se a posição zero, ignorada neste programa). Se houver, pelo menos, uma posição em que haja um espaço e não uma jogada, a função retorna `False`.

O loop `for` desta função verificará todas as posições do tabuleiro e, se ele for até o fim, quer dizer que todos as posições estão ocupadas, caracterizando a "velha" (o empate). Se houver alguma posição vazia, o loop é interrompido e retorna `False`.

7.15 O início do jogo

O jogo inicia, de fato, na linha 140. É feita uma saldação ao jogador e uma lista com 10 posições, cada uma possuindo um espaço, é criada.

Em seguida, é pedido que o jogador escolha o símbolo que o representará no jogo. O símbolo do jogador e do computador são designados às variáveis `playerLetter` e `computerLetter` por múltipla atribuição, com a função `inputPlayerLetter()`.

Então, é decidido quem joga primeiro, na linha 145, com a função `whoGoesFirst()`.

7.15.1 Rodando a vez do jogador

Na linha 150, o jogo começa, em um loop `while`, e permanece neste loop até que seja finalizado. Para controlar isto, é criada a variável `gameIsPlaying` e é designada o valor `True` a ela. A função `whoGoesFirst()` retorna `'player'` ou `'computer'`. No caso de ela retornar `'player'`, continua a execução na linha 151. Do contrário, irá para a linha 169.

Se for a vez do jogador, a primeira coisa feita é mostrar o tabuleiro, chamando a função `drewBoard()` e passando a variável `theBoard`. Em seguida, é pedido que o jogador digite a sua jogada, através da função `getPlayerMove()` e a jogada é feita, de fato, ao chamar a função `makeMove()`.

Agora que o jogador fez a sua jogada, o programa verificará se ele ganhou o jogo. Se `isWinner()` retorna `True`, o tabuleiro da vitória deve ser mostrado, seguido de uma mensagem de que o jogador venceu. Consequentemente, `gameIsPlaying` recebe `False` e o jogo termina.

Pelo contrário, se o jogador não venceu o jogo, verifica-se se o tabuleiro está cheio e deu empate, pois o movimento do jogador pode ter ocupado a última casa do tabuleiro e não haver vencedores. Esta condição é verificada pela função `isBoardFull()`.

7.15.2 Rodando a vez do computador

Se nenhuma das opções anteriores terminar o jogo, então é vez do computador e a variável `turn` recebe `'computer'`. O código é bem similar ao movimento do jogador, entretanto, não há a necessidade de mostrar o jogo na tela.

Ainda, se o jogo não for encerrado, é a vez do jogador e o processo se repete até o fim do jogo.

7.15.3 Finalizando o jogo

No fim do bloco `while` que caracteriza a execução do jogo, há um `if` para o jogador escolher jogar novamente. Esta escolha é permitida pela função `playAgain()` e, se o jogador desejar, o loop `while` que inicia o jogo continua executando. Senão, o loop é interrompido e o programa termina.

7.16 Exercícios complementares

1. **Verificação de CPF**. Desenvolva um programa que solicite a digitação de um número de CPF no formato $xxx.xxx.xxx - xx$ e indique se é um número válido ou inválido através da validação dos dígitos verificadores e dos caracteres de formatação. Implemente uma função para fazer cada verificação. Dica: faça uma pesquisa na internet para entender como é verificado se um CPF é válido ou não.
2. (Adaptada de ACM ICPC) - **Damas do Xadrez**. Desenvolva um programa que dado duas posições em um jogo de xadrez retorne quantas jogadas é preciso para uma dama se mover da primeira para a segunda posição.
3. Crie um programa que dada um sequencia de n números ordene em ordem decrescente. Obs.: Você deverá criar a função para ordenar os números, e não utilizar funções prontas de ordenação.
4. **Quadrado mágico**. Um quadrado mágico é aquele dividido em linhas e colunas, com um número em cada posição e no qual a soma das linhas, colunas e diagonais é a mesma. Por exemplo, veja um quadrado mágico de lado 3, com números de 1 a 9:
   8 3 4
   1 5 9
   6 7 2
   Elabore um programa que receba como entrada um quadrado $3x3$ e retorne se ele é ou não um quadrado mágico.
5. Adapte exercício anterior para receber qualquer quadrado nxn, o valor de n deve ser o primeiro valor fornecido ao programa.

8 Bagels

Tópicos abordados neste capítulo:

- Operadores compostos, $+=$, $-=$, $*=$, $/=$
- A função `random.shuffle()`
- Os métodos de lista `sort()` e `join()`
- Interpolação (formatação) de strings
- Especificador de conversão `%s`
- Loops aninhados

Neste capítulo, serão apresentadas mais funções do Python. Além disso, duas formas de simplificar algumas operações são apresentadas: formatação de strings e operadores compostos.

Bagels é um jogo simples: um número de três dígitos sem repetição é sorteado é o jogador tenta adivinhar qual é este número. Após cada palpite, uma dica é dada sobre tal número, relacionada com o tal palpite. Se a dica for *bagels*, quer dizer que nenhum algarismo está correto; se for *pico*, então um dos algarismos está correto, mas no lugar errado; e, finalmente, de for *fermi*, o palpite possui um algarismo certo no lugar certo.

O jogo também fornece múltiplas dicas depois de cada palpite. Se o número for 456 e o palpite é 546, o palpite será "fermi pico pico", pois um dígito está correto e no lugar certo também, mas dois dígitos estão em lugares errados.

8.1 Modelo de execução

```
I am thinking of a 3-digit number. Try to guess what it is.
Here are some clues:
When I say:    That means:
  Pico         One digit is correct but in the wrong position.
  Fermi        One digit is correct and in the right position.
  Bagels       No digit is correct.
I have thought up a number. You have 10 guesses to get it.
Guess #1:
753
Pico
Guess #2:
890
Bagels
Guess #3:
764
Fermi
Guess #4:
524
Bagels
Guess #5:
364
Fermi Fermi
Guess #6:
314
Fermi Pico
Guess #7:
361
You got it!
Do you want to play again? (yes or no)
no
```

8.2 Código-fonte do jogo

Código 6: Bagels.

```
1  import random
2  def getSecretNum(numDigits):
3      # Retorna uma string do tamanho numDigits, com algarismos diferentes.
4      numbers = list(range(10))
5      random.shuffle(numbers)
6      secretNum = ''
7      for i in range(numDigits):
8          secretNum += str(numbers[i])
9      return secretNum
10
11 def getClues(guess, secretNum):
12     # Retorna uma string com as dicas pico, fermi, bagels ao usuario.
13     if guess == secretNum:
14         return 'You got it!'
15
16     clue = []
17
18     for i in range(len(guess)):
19         if guess[i] == secretNum[i]:
20             clue.append('Fermi')
21         elif guess[i] in secretNum:
22             clue.append('Pico')
23     if len(clue) == 0:
24         return 'Bagels'
25
26     clue.sort()
27     return ' '.join(clue)
28
29 def isOnlyDigits(num):
30     # Retorna True se num for uma string. Do contrario, retorna False.
31     if num == '':
32         return False
33
34     for i in num:
35         if i not in '0 1 2 3 4 5 6 7 8 9'.split():
36             return False
37
38     return True
39
40 def playAgain():
41     # Esta funcao retorna True se o jogador deseja jogar novamente.
42     print('Do you want to play again? (yes or no)')
43     return input().lower().startswith('y')
44
45 NUMDIGITS = 3
46 MAXGUESS = 10
47
48 print('I am thinking of a %s-digit number. Try to guess what it is.' % (
       NUMDIGITS))
49 print('Here are some clues:')
50 print('When I say:    That means:')
51 print('  Pico         One digit is correct but in the wrong position.')
52 print('  Fermi        One digit is correct and in the right position.')
53 print('  Bagels       No digit is correct.')
54
55 while True:
56     secretNum = getSecretNum(NUMDIGITS)
```

```
57      print('I have thought up a number. You have %s guesses to get it.' % (
            MAXGUESS))
58
59      numGuesses = 1
60      while numGuesses <= MAXGUESS:
61          guess = ''
62          while len(guess) != NUMDIGITS or not isOnlyDigits(guess):
63              print('Guess #%s: ' % (numGuesses))
64              guess = input()
65
66          clue = getClues(guess, secretNum)
67          print(clue)
68          numGuesses += 1
69
70          if guess == secretNum:
71              break
72          if numGuesses > MAXGUESS:
73              print('You ran out of guesses. The answer was %s.' % (secretNum))
74
75      if not playAgain():
76          break
```

8.3 Projeto do jogo

O fluxograma que caracteriza o funcionamento do jogo está ilustrado na figura 15.

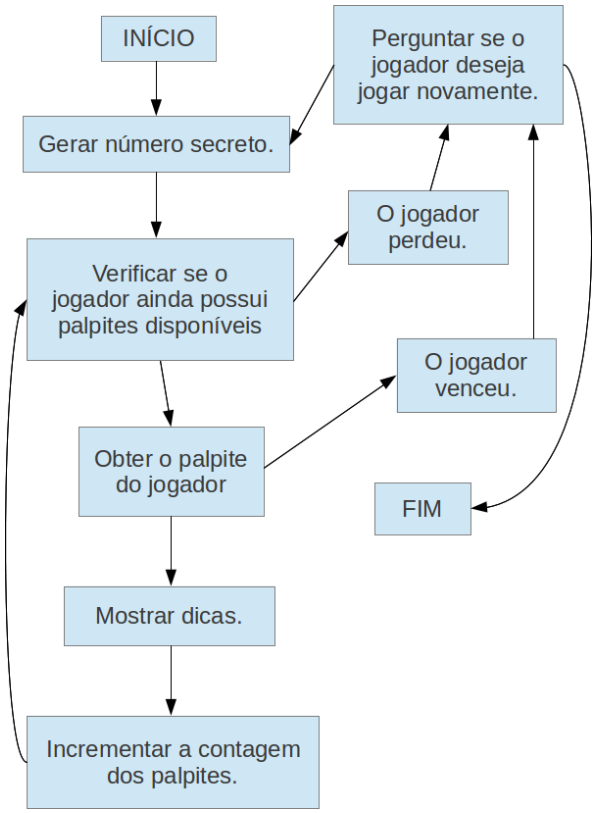

Figura 15: Fluxograma para o jogo Bagels.

8.4 Como o programa funciona

No início do programa, chamamos o módulo `random` e criamos uma função para gerar um número aleatório para o jogador adivinhar. A função `getSecretNum()` gera este número ao invés do código produzir apenas um número de três dígitos, utilizamos o parâmetro `numDigits` para estabelecer quantos serão estes dígitos (desta forma, podemos gerar números com 4 ou 6 dígitos, por exemplo).

Para tal, é utilizada função `range()` com a função `list()` e, assim, uma lista de inteiros de 0 a 9 é criada. Em seguida, a função `random.shuffle()` é chamado e a referência a esta lista é passada por parâmetro. Esta função não retorna valor algum, ela apenas muda os elementos de lugar.

O número secreto será dado pelos três primeiros números da lista embaralhada (pois `numDigits` vale 3, neste jogo). Este número será armazenado na string `secretNum`, que inicia vazia, e será o retorno da função. Observe que este retorno não é um número, mas uma string.

8.4.1 Operadores compostos

Há um novo operador na linha 8, o + =. Este tipo de operação é chamada de operação composta, e é utilizada para incrementar uma variável ou concatenar strings. Neste programa, os três primeiros números da lista embaralhada são concatenados através do loop `for`.

Uma maneira frequentemente utilizada para incremento de variáveis, por exemplo, em um loop, é atribuir o valor de uma variável a ela mesma, mais um certo valor. Observe:

```
>>> var1 = 3
>>> var1 += 1  # esta linha é o mesmo que var1 = var1 + 1
>>> var1
4
```

Também é possível utilizar este formato de operação para outras além da soma, como a subtração −=, multiplicação ⋆= e divisão /=.

8.4.2 Retornando a dica ao jogador

A função `getClues()` dará a dica em forma de string (podendo ser mais de uma) para o jogador, dependendo do que ela recebe como argumento, através dos parâmetros `guess` `secretNum`. O passo mais óbvio (e o primeiro da verificação) é saber se o palpite do jogador está correto.

Se o palpite não for o número sorteado, é necessária a verificação de cada algarismo, para dar a(s) dica(s) ao jogador. Para tal, armazenamos as dicas em uma lista `clue` e é feita a comparação através da entrada do jogador no laço `for` que inicia na linha 18.

Caso não haja nenhum dígito correto no palpite do jogador, então a lista `clue` estará vazia e esta é a condição para que seja impresso `'Bagels'`, no bloco `if` da linha 23.

8.4.3 O método de lista `sort()`

Na linha 26, a lista `clue` chama o método `sort()`. Este método ordena os elementos em ordem alfabética, não retornando uma nova lista ordenada, mas ordenando-os na própria lista. A intenção de ordenar as dicas passadas para o jogador é não indicar a qual dígito cada dica pertence, tornando o jogo mais difícil.

8.4.4 O método de string `join()`

O retorno dado na linha 27 possui uma string contendo um espaço e chamando o método `join()`, que concatena as strings de uma lista com a string que chamou o método. Ou seja, as dicas da lista `clue` serão agrupadas em uma mesma string com um espaço entre cada elemento (dica).

8.4.5 Verificando se a string possui apenas números

A função `isOnlydigits()` verifica se a entrada do jogador possui apenas números. Ela retorna `False` se o palpite não houver 3 algarismos e mantém-se pedindo ao jogador um novo palpite, até que este seja válido.

8.5 Início do jogo

O jogo inicia a ser executado, de fato, na linha 45. A quantidade de dígitos do número adivinhado assim como a quantidade de palpites possíveis são definidos pelas constantes `NUMDIGITS` e `MAXGUESS`.

8.5.1 Interpolação de strings

Interpolação/formatação de strings é outro atalho para operações, mas, desta vez, com strings. Normalmente, quando é necessário concatenar uma string armazenada em uma variável, utiliza-se o operador +. Entretanto, isto pode não ser conveniente ao utilizar muitas variáveis a serem impressas. Portanto, há uma forma mais sucinta de lidar com muitas variáveis que carregam strings, utilizando o "substituto" `%s` indicando onde cada variável deva ser impressa. Observe o exemplo:

```
>>> name = 'Alice'
>>> event = 'party'
>>> where = 'the pool'
>>> when = 'Saturday'
>>> time = '6:00pm'
>>> print('Hello, %s. Will you go to the %s at %s this
%s at %s?' % (name, event, where, when, time))
Hello, Alice. Will you go to the party at the pool
this Saturday at 6:00pm?
>>>
```

Estes substitutos são chamados de **especificadores de conversão** e permitem que as variáveis sejam colocadas todas de uma vez no fim de uma sentença. Outra vantagem desta ferramenta é que ela pode ser utilizada com qualquer tipo de variável, não apenas strings:

```
>>> var1 = 76
>>> print('var 1 == %s' %(var1))
var 1 == 76
```

8.5.2 Utilizando as funções descritas e verificando a vitória do jogador

O código a partir da linha 55 é similar ao dos programas anteriores, mantendo o jogo rodando dentro de um loop. Neste loop, as funções para criar um número e obter o palpite do jogador são utilizadas, assim como a função para dar as dicas ao jogador, nesta ordem.

Para verificar se o jogador venceu o jogo ou não, são verificadas as condições nos blocos `if` iniciados na linha 70. A verificação óbvia é se o número sorteado é o mesmo que o jogador deu como entrada, se for, então o jogador vence. Outra alternativa é o jogador esgotar os palpites e perder, condição verificada no segundo `if`, da linha 72.

As outras funcionalidades do programa são similares ao conteúdo que já foi estudado. Para esclarecer como o programa funciona, se ainda for necessário, faça o teste de mesa do programa. **Teste de mesa** de um programa é quando o programador verifica passo a passo, cuidadosamente, o programa, supondo entradas do usuário e "rodando o programa no papel". Esta prática é muito útil para encontrar erros no programa (ou em trechos dele) e entender, de fato, como um código funciona.

Tarefa 8.1

Pesquise, para a próxima aula, sobre os seguinte métodos:

- Listas bidimensionais (listas de listas) e aplicações.
- O método de lista `remove()`
- O método de string `isdigit()`
- A função `sys.exit()`

Tarefa 8.2

Pesquise sobre coordenadas cartesianas e números negativos. Relacione esta pesquisa ao desenvolvimento de jogos (mesmo que seja de jogos simples, como os desenvolvidos nesta oficina). Procure ter afinidade com estes tópicos, pois eles serão frequentemente utilizados em jogos com interface gráfica.

1. Esta pesquisa poderá ser entregue por e-mail ou na forma impressa. No caso do e-mail, através de um documento em anexo em formato .doc ou .pdf, de preferência pdf.
2. Entrega: no máximo, 15 dias após esta aula.
3. Formato: fonte 12pt, preferencialmente Arial, Times New Roman ou similares. Trabalhos em Comic Sans e outras fontes informais ou estranhas não serão aceitos.
4. Conteúdo: mínimo duas, máximo cinco páginas, não é necessário ter capa/folha de rosto/etc., é permitido que seja feito em forma de artigo.
5. O trabalho deverá ser devidamente referenciado e Wikipedia ou uma única referência (ou seja, copy/paste) não serão aceitas.

9 O Criptograma de César

Tópicos abordados neste capítulo:

- Criptografia e criptogramas
- Encriptação e decodificação de mensagens
- Valores ASCII
- Funções `chr()` e `ord()`
- Os métodos de string `isalpha()`, `isupper()` e `islower()`
- Método de força bruta

O programa deste capítulo não é bem um jogo, mas é divertido, de qualquer forma. Este programa converterá uma frase em um código secreto e converterá códigos secretos de volta à frase original. Apenas quem souber como o código secreto é composto irá decodificar a mensagem.

Através destes procedimentos, serão abordados novos métodos e funções para lidar com strings. Além disso, será apresentado como lidar com aritmética envolvendo strings.

9.1 Criptografia

A ciência de codificar mensagens é chamada de criptografia. Ela tem sido utilizada por milhares de anos, para enviar mensagens que apenas o destinatário poderá entender. O sistema utilizado para codificar uma mensagem é chamado de **criptograma**.

Em criptografia, a mensagem a ser criptografada é chamada de texto puro. Este texto é convertido em uma mensagem encriptada. Neste programa, a mensagem encriptada parecerá uma série de letras aleatórias, e não será possivel entendê-lo da mesma forma que a mensagem original.

Entretanto, se a criptografia da mensagem for conhecida, é possível decodificá-la para obter o texto original.

Muitas criptografias utilizam chaves. **Chaves** são valores secretos que permitem decodificar uma mensagem que foi encriptada utilizando um sistema particular.

9.2 A encriptação de César

Ao encriptar uma mensagem, utiliza-se uma chave para encriptá-la ou desencriptá-la. A chave para o programa desenvolvido aqui será um número entre 1 e 26. Somente sabendo a chave correta é possível decodificar a mensagem.

O criptograma de César é um dos mais antigos já inventados. Neste criptograma, cada letra é substituída por um caractere à frente de acordo com a ordem do alfabeto. Em criptografia, estas letras são chamadas de **símbolos**, pois podem ser letras, números ou qualquer outra coisa. Observe um exemplo de como funcionaria a correspondência com a chave 3 na figura 16.

Figura 16: Alfabeto substituído por caracteres 3 casas à frente.

9.3 ASCII, utilizando números para localizar letras

Como implementar o avanço dos caracteres neste programa? Isto é feito representando cada letra com um número (ordinal) e então somando ou subtraindo este número para formar um novo, correspondendo à uma nova letra, consequentemente. O código ASCII faz a correspondência para cada caracter em um número entre 32 e 127. Os caracteres que correspondem aos números menores que 32 são "não imprimíveis", logo, não serão utilizados.

Tabela 7: Tabela ASCII.

32	(espaço)	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	∧	110	n	126	∼
47	/	63	?	79	O	95	_	111	o		

As letras maiúsculas, de A a Z, são representadas pelos números 65 a 90. As minúsculas, de a a z, são representadas pelos números entre 97 e 122. Os dígitos de 0 a 9 são representados pelos números entre 48 e 57.

Observe a tabela 7. Se quisermos avançar 3 casas para a letra A, por exemplo, converte-se no número 65. Então, $65 + 3 = 68$, gerando a letra D. Usaremos as funções `chr()` e `ord()` para conversão entre letras e números.

9.4 As funções `chr()` `ord()`

A função `chr()` (de “char”, versão curta de “character”) obtém o caracter ASCII para o parâmetro, que deve ser um número. A função `ord()` (de “ordinal”) faz o inverso, retorna um valor inteiro, correspondente ao valor ASCII do caracter passado por parâmetro.

9.5 Modelo de execução do programa

```
Do you wish to encrypt or decrypt a message?
encrypt
Enter your message:
The sky above the port was the color of television, tuned to adead channel.
Enter the key number (1-26)
13
Your translated text is:
Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb n
qrnq punaary.
Now we will run the program and decrypt the text that we just
encrypted.
Do you wish to encrypt or decrypt a message?
decrypt
Enter your message:
Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb nqrnq punaary.
Enter the key number (1-26)
13
Your translated text is:
```

```
The sky above the port was the color of television, tuned to a
dead channel.
```

Observe o que ocorre se for dada a chave errada:

```
Do you wish to encrypt or decrypt a message?
decrypt
Enter your message:
Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb nqrnq punaary.
Enter the key number (1-26)
15
Your translated text is:
Rfc qiw yzmtc rfc nmpr uyq rfc amjmp md rcjctgqgml, rslcb rm y
bcyb afyllcj.
```

9.6 Código-fonte

Código 7: Criptograma de César.

```
1  # Caesar Cipher
2
3  MAX_KEY_SIZE = 26
4
5  def getMode():
6      while True:
7          print('Do you wish to encrypt or decrypt a message?')
8          mode = input().lower()
9          if mode in 'encrypt e decrypt d'.split():
10             return mode
11         else:
12             print('Enter either "encrypt" or "e" or "decrypt" or "d".')
13
14 def getMessage():
15     print('Enter your message:')
16     return input()
17
18 def getKey():
19     key = 0
20     while True:
21         print('Enter the key number (1-%s)' % (MAX_KEY_SIZE))
22         key = int(input())
23         if (key >= 1 and key <= MAX_KEY_SIZE):
24             return key
25
26 def getTranslatedMessage(mode, message, key):
27     if mode[0] == 'd':
28         key = -key
29     translated = ''
30
31     for symbol in message:
32         if symbol.isalpha():
33             num = ord(symbol)
34             num += key
35
36             if symbol.isupper():
37                 if num > ord('Z'):
38                     num -= 26
39                 elif num < ord('A'):
40                     num += 26
41             elif symbol.islower():
```

```
42                  if num > ord('z'):
43                      num -= 26
44                  elif num < ord('a'):
45                      num += 26
46
47              translated += chr(num)
48          else:
49              translated += symbol
50      return translated
51
52  mode = getMode()
53  message = getMessage()
54  key = getKey()
55
56  print('Your translated text is:')
57  print(getTranslatedMessage(mode, message, key))
```

9.7 Como o código funciona

Este código é bem mais curto do que os anteriores, entretanto, os conceitos abordados neste capítulo são muito úteis no desenvolvimento de jogos. Observemos os tópicos principais deste programa.

9.7.1 Decidindo se o usuário quer encriptar ou desencriptar uma mensagem

A função `getMode()` permite que o usuário faça a escolha entre encriptar ou desencriptar/decodificar uma mensagem. Ela retorna a resposta do jogador, obtida através da comparação da entrada do usuário com a lista gerada por `'encrypt e decrypt d'.split()`.

9.7.2 Obtendo a mensagem do jogador

A função `getMessage()` obtém a mensagem a ser codificada ou decodificada e a retorna.

9.7.3 Obtendo a chave do jogador

A função `getKey()` permite ao jogador digitar a chave que permitirá encriptar ou desencriptar a mensagem que será fornecida. O loop `while`, da linha 20, assegura que a função retorne uma chave válida, que é um número entre 1 e 26. Desta forma, a função retorna um inteiro.

9.7.4 Encriptando ou desencriptando a mensagem com a chave dada

`getTranslatedMessage()` é a função que faz a encriptação ou desencriptação no programa. Ela possui três parâmetros:

- `mode`: define se a função irá encriptar ou desencriptar a mensagem.
- `message`: mensagem a ser encriptada ou desencriptada
- `key`: chave utilizada na (de)codificação

A primeira linha da função determina o modo de trabalho da função, comparando a primeira letra da entrada do jogador e, neste caso, mudar o sinal da chave para fazer o processo inverso (que seria a encritpação).

`translated` é a string que armazenará o resultado final: ou a mensagem encriptada ou a desencriptada. Esta variável apenas recebe strings concatenadas, iniciando por uma string vazia.

9.7.5 Encriptando e desencriptando cada letra

Este processo é bem simples e o mesmo bloco de código é utilizado para a conversão de cada caracter. O código a partir da linha 36 verifica se a string possui algum caracter em letras maiúsculas.

Se houver, há duas situações para se preocupar: se a letra é Z (ou seja, a última letra do alfabeto) e o deslocamento resultar em um caracter que não seja uma letra; e no caso de a letra for A, que o deslocamento deverá ser maior que o número ASCII deste caracter. O mesmo ocorre para mensagens com 'a' e 'z'.

Se o programa estiver no modo de desencriptação, a chave será negativa, assim, o processo inverso ocorrerá. A string `translated` obterá os caracteres de cada encriptação ou desencriptação. Se o símbolo não for uma letra, o código da linha 48 executará e o símbolo original aparecerá na string transformada. Isto significa que outros caracteres não passarão pelo processo.

A última linha da função `getTranslatedMessage()` retorna a string transformada.

9.7.6 O início do programa

O programa inicia a execução na linha 52. Cada função definida é chamada e armazenada em uma variável, pois elas possuem valores de retorno. Então, estes valores são passados para a função `getTranslatedMessage()`, que retorna o valor a ser impresso para o usuário.

9.8 O método de string `isalpha()`

Este método retornará `True` se a string for inteiramente composta de maiúsculas ou minúsculas de A a Z, sem espaços ou se não for uma string em branco. Se a string contém quaisquer caracteres que não sejam letras, então o retorno é `False`.

9.9 Os métodos de string `isupper()` e `islower()`

Os métodos `isupper()` e `islower()` funcionam de forma similar aos métodos `isdigit()` e `isalpha()`. `isupper()` retornará `True` se a string a qual o chama possuir pelo menos uma letra maiúscula e nenhuma letra minúscula. O método `islower()` funciona ao contrário: retorna `True` se a string possuir apenas letras minúsculas. Se as condições não forem satisfeitas, o retorno é `False`. A existência de caracteres que não sejam letras não afetam a saída, ou seja, os métodos analisam apenas as letras de uma string.

9.10 Força Bruta

Embora o programa possa enganar algumas pessoas e esconder as mensagens, o código pode ser quebrado através de análise criptográfica. Um dos métodos de desencriptação sem saber a chave (ou o código) por trás de uma mensagem é a força bruta. **Força bruta** é a técnica de tentar todo procedimento possível (no caso deste programa, testar todas as chaves) até encontrar o resultado correto. O número de chaves deste programa é relativamente pequeno e pode-se gerar todos os resultados possíveis até encontrar algum que faça sentido. Vamos adicionar um método de força bruta ao programa.

9.10.1 Adicionando um método de força bruta ao programa

Observe as alterações na função `getMode()`:

```
5  def getMode():
6      while True:
7          print('Do you wish to encrypt or decrypt or brute force a message?')
8          mode = input().lower()
9          if mode in 'encrypt e decrypt d brute b'.split():
10             return mode[0]
11         else:
12             print('Enter either "encrypt" or "e" or "decrypt" or "d"
13                 or "brute" or "b".')
```

Estas alterações permitem que a força bruta seja uma opção do programa. Então, adicione e modifique as seguintes linhas do programa:

```
5  mode = getMode()
6  message = getMessage()
7  if mode[0] != 'b':
8      key = getKey()
9
10 print('Your translated text is:')
11 if mode[0] != 'b':
12     print(getTranslatedMessage(mode, message, key))
13 else:
14     for key in range(1, MAX_KEY_SIZE + 1):
15         print(key, getTranslatedMessage('decrypt', message, key))
```

Estas mudanças fazem o seu programa pedir ao usuário uma chave, se não estiver no modo de força bruta. Se for o caso, então a função `getTranslatedMessage()` funcionará da mesma forma que anteriormente.

Entretanto, se o modo força bruta for acionado, então a função `getTranslatedMessage()` executará um loop iterando de 1 a `MAX_KEY_SIZE` (que é 26). Este programa imprimirá todas as "traduções" possíveis de uma mensagem.

9.11 Exercícios complementares

1. Explique apenas as alterações que você faria no programa *Caesar Cipher* para que:
   - Não fosse necessário fazer as comparações das linhas 37, 39, 42 e 44 para que as operações não ultrapassem o limite de letras da tabela ASCII. Dica: Use o operador de módulo.
   - Usando suas alterações do exercício acima (economizará algumas linhas de código) implemente um método alternativo e mais seguro de criptografia e descriptografia neste programa. Para cada posição da string que será criptografada:
     - Calcule o resto da divisão do índice por quatro
     - Caso o resto seja 0, adicione 10 ao valor numérico do caractere
     - Caso o resto seja 1, subtraia 10 do valor numérico do caractere
     - Caso o resto seja 2, adicione 20 ao valor numérico do caractere
     - Caso o resto seja 3, subtraia 20 do valor numérico do caractere
     - Adicione o valor do índice ao valor numerico do caractere
   - Crie um método próprio para criptografar e descriptografar a string

10 Introdução à Gráficos e Animação (parte 1)

Tópicos abordados neste capítulo:

- Bibliotecas de software
- Pygame
- Interfaces gráficas de usuário (GUI: Graphical User Interfaces)
- Desenhos primitivos
- Criando uma janela GUI com o Pygame
- Cores em Pygame
- Fontes em Pygame
- Gráficos linearizados
- Atributos
- Os tipos de dados `pygame.font.Font`, `pygame.Surface`, `pygame.Rect`, `pygame.PixelArray`
- Construtores
- A função `type()`
- Funções de desenho do Pygame
- O método `blit()` para objetos `Surface`
- Eventos
- O loop de um jogo

Até agora, os jogos e programas trabalhados utilizavam apenas modo texto: um texto era mostrado na tela, como saída, e o jogador digitava algo do teclado (outro texto). Tudo isto serviu para introduzir os conceitos fundamentais de programação. A partir desta aula, veremos como utilizar recursos gráficos através da biblioteca Pygame.

Nas próximas aulas, o conceito da lógica de um jogo será mais simples do que foi visto até agora, mas todas as ferramentas apresentadas poderão ser utilizadas com lógicas mais robustas, como o jogo da velha ou da forca.

Uma biblioteca permite ao programador adicionar mais recursos ao seu programa, de maneira genérica, mas customizável, sem precisar se preocupar em programar todo o código. O Pygame é uma biblioteca que possui módulos para gráficos, sons e outros recursos comumente utilizado para a programação de jogos.

10.1 Pygame

O Pygame é uma biblioteca de software que não faz parte do Python "nativo". Mas, assim como o Python, essa biblioteca está disponível gratuitamente, multiplataforma. Assegure-se que a versão que você for baixar seja compatível com o Python 3, utilizado nesta oficina.

10.2 Hello World

Novamente, criaremos um "Hello World", como no início do curso. Entretanto, adicionaremos recursos gráficos, como uma janela de **interface gráfica para usuário** (GUI). Uma interface gráfica permite que sejam criadas janelas, cores, formas e imagens que podem ser desenhadas pelo programa, assim como aceitar entradas pelo mouse (e não apenas pelo teclado). As formas básicas são chamadas de **desenhos primitivos**. Além disso, janelas são utilizadas ao invés do terminal.

O Pygame não trabalha muito bem com o terminal pois ele exige um *loop de jogo* (que será explicado posteriormente). Devido a isto, não é possível enviar comandos para o Pygame através do terminal interativo.

Além disso, programas do Pygame não utilizam a função `input()`. Não há texto de entrada ou saída. Ao invés disso, as saídas do programa são dadas através de uma janela. As entradas são captadas através do mouse ou do teclado por **eventos**, que serão vistos na próxima aula. Porém, se o programa tiver erros, eles serão mostrados na linha de comando.

Pode-se usar a função `print()`, entretanto, a mensagem aparecerá na linha de comando, sendo ineficiente para saída, mas inteligente para debug. Por exemplo, o resultado de uma operação pode ser impresso para o programador saber se aquele trecho do programa está funcionando corretamente[21].

10.3 Código-fonte do Hello World

Iniciaremos com um programa bem simples, que mostra texto e algumas formas em uma janela. Execute o código 8.

Código 8: Hello World gráfico.

```
1  import pygame, sys
2  from pygame.locals import *
3
4  # inicia o pygame
5  pygame.init()
6
7  # inicia a janela
8  windowSurface = pygame.display.set_mode((500, 400), 0, 32)
9  pygame.display.set_caption('Hello world!')
10
11 # inicia as cores utilizadas
12 BLACK = (0, 0, 0)
13 WHITE = (255, 255, 255)
14 RED = (255, 0, 0)
15 GREEN = (0, 255, 0)
16 BLUE = (0, 0, 255)
17
18 # inicia as fontes
19 basicFont = pygame.font.SysFont(None, 48)
20
21 # inicia o texto
22 text = basicFont.render('Hello world!', True, WHITE, BLUE)
23 textRect = text.get_rect()
24 textRect.centerx = windowSurface.get_rect().centerx
25 textRect.centery = windowSurface.get_rect().centery
26
27 # desenha o fundo branco
28 windowSurface.fill(WHITE)
29
30 # desenha um poligono verde na superficie
31 pygame.draw.polygon(windowSurface, GREEN, ((146, 0), (291, 106), (236, 277),
       (56, 277), (0, 106)))
32
33 # desenha algumas linhas azuis na superficie
34 pygame.draw.line(windowSurface, BLUE, (60, 60), (120, 60), 4)
35 pygame.draw.line(windowSurface, BLUE, (120, 60), (60, 120))
36 pygame.draw.line(windowSurface, BLUE, (60, 120), (120, 120), 4)
37
38 # desenha um circulo azul na superficie
39 pygame.draw.circle(windowSurface, BLUE, (300, 50), 20, 0)
40
41 # desenha uma elipse vermelha na superficie
42 pygame.draw.ellipse(windowSurface, RED, (300, 250, 40, 80), 1)
43
44 # desenha o retangulo do fundo do texto na superficie
45 pygame.draw.rect(windowSurface, RED, (textRect.left - 20, textRect.top - 20,
       textRect.width + 40, textRect.height + 40))
46
```

[21] Você pode consultar informações de como utilizar o Pygame através do site do desenvolvedor: http://pygame.org/docs/ref/.

```
47 # obtem um array de pixel da superficie
48 pixArray = pygame.PixelArray(windowSurface)
49 pixArray[480][380] = BLACK
50 del pixArray
51
52 # desenha o texto na janela
53 windowSurface.blit(text, textRect)
54
55 # desenha a janela na tela
56 pygame.display.update()
57
58 # roda o loop do jogo
59 while True:
60     for event in pygame.event.get():
61         if event.type == QUIT:
62             pygame.quit()
63             sys.exit()
```

A execução deste programa resultará em uma janela similar à figura 17.

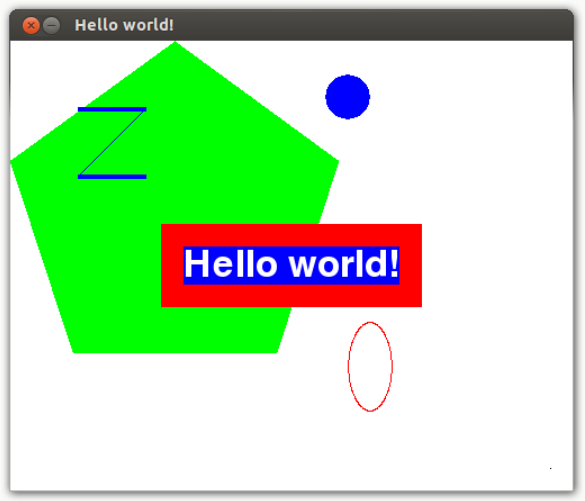

Figura 17: Resultado da execução do código 8.

O interessante em utilizar uma GUI ao invés do console, é que o texto pode aparecer em qualquer lugar da janela, não apenas depois do último texto impresso. Além disso, o texto pode ter qualquer cor ou tamanho. O Pygame utiliza muitas tuplas, estruturas parecidas com listas, mas com parênteses delimitadores ao invés de colchetes. A maior diferença entre tuplas e listas é que não é possível alterar, adicionar ou remover valores em uma tupla. Por razões técnicas, sabendo que os valores não podem ser alterados em uma tupla, permite-se ao Python que trabalhe de forma mais eficiente.

10.4 Importando o módulo Pygame

Na primeira linha do programa, os módulos `pygame` e `sys` são importados para que seja permitido utilizar suas funções. Pode-se chamar mais de uma módulo em uma mesma sentença, separados por vírgulas.

A segunda linha importa o módulo `pygame.locals`. Este módulo contém inúmeras constantes, como `QUIT` e `K_ESCAPE`, explicadas posteriormente. Usando a forma `from nomeDoMódulo import *` faz com que não seja necessário digitar `pygame.locals` antes de cada função deste módulo, como vêm sendo feito desde os primeiros programas. O asterisco da sentença significa que todo o conteúdo do módulo é importado para o programa. Se o módulo `sys` fosse importado da forma `from sys import *`, ao invés de `import sys`, seria possível chamar apenas a função `exit()` ao invés de `sys.exit()`. Entretanto, é indicado que se use o nome dos módulos para que o programador saiba a qual deles pertencem as funções do programa.

10.5 Funções e métodos do programa

10.5.1 `pygame.init()`

O software da biblioteca Pygame possui um código inicial que precisa ser executado antes de utilizá-la. Todos os programas que utilizem o Pygame devem executar este código através da linha `pygame.init()`, logo depois de ter importado o módulo `pygame`, mas antes de chamar as funções deste módulo.

10.5.2 pygame.display.set_mode()

A linha 8 cria uma janela GUI, para o programa, chamando o método `set_mode()` do módulo `pygame.display`. (O módulo `display` é um módulo dentro do módulo `pygame`.)

O método `set_mode()` exige três parâmetros: o primeiro é uma tupla de dois inteiros para a largura e altura da janela, em pixels. Um **pixel** é o menor ponto de uma tela de computador. Um único pixel pode ser de qualquer cor. Todos os pixels da tela trabalham mutuamente para gerar as imagens. Utilizou-se a tupla `(500, 400)` para o primeiro parâmetro do método `set_mode()` para estabelecer a dimensão da janela.

O segundo e o terceiro parâmetros são para opções avançadas da GUI. Como, por enquanto, estes parâmetros são desnecessários, os valores zero e 32 foram passados, pois o método não pode receber menos parâmetros do que o esperado.

O método `set_caption()` retorna um objeto `pygame.Surface` (que serão denominados como objetos `Surface`, para simplificar) com texto. Pode-se armazenar objetos em variáveis, como qualquer outro valor. O objeto `Surface` representa a janela e a variável `windowSurface` deverá ser inclusa em todas as chamadas de função de desenho.

10.6 Referências de objetos

Variáveis que "guardam" objetos, na verdade, funcionam como variáveis de referência à listas, vistas anteriormente. Diferentemente das variáveis "comuns", ou seja, que podem fazer a cópia de um valor e, assim, mantê-lo armazenado, as variáveis as quais possuem objetos atribuídos apenas levam uma referência deste objeto. Logo, se duas variáveis referenciarem um objeto e apenas uma delas sofrer alteração, automaticamente a outra também será alterada, pois o valor (ou conjunto de valores) que elas carregam é o mesmo.

10.7 Cores em Pygame

As linhas 12 a 16 do código 8 possuem valores declarados como tuplas de três elementos, com valores que identificam cores em RGB. Cada valor dos parênteses se refere à "quantidade" da cor em vermelho (R, red), verde (G, green) e azul (B, blue), respectivamente.

A tabela 8 possui algumas combinações RGB de cores comuns.

No código 8, as variáveis receberam nomes em maiúsculas para indicar que são constantes, ou seja, não mudarão ao longo do programa. Desta forma, fica mais fácil referenciar a cor preta por `BLACK` ao invés de sempre digitar `(0, 0, 0)`.

10.8 Fontes

Uma **fonte** é um conjunto de letras, números e símbolos diversos que seguem em estilo estabelecido. Nos programas em linha de comando, não é possível estabelecer uma fonte própria para o programa, apenas para todo o console. Entretanto, é possível desenhar letras diferentes através de uma GUI, dizendo ao Pygame onde e como desenhar o gráfico desejado.

10.8.1 A função `pygame.font.SysFont()`

Na linha 19, um objeto `pygame.font.Font` é criado (que será chamado de objeto `Font`, para facilitar) através da chamada de função `pygame.font.SysFont()`. O primeiro parâmetro é o

Tabela 8: Cores RGB.

Cor	Valores RGB
Amarelo	(255, 255, 0)
Azul	(0, 0, 255)
Azul claro (cornflower)	(100, 149, 237)
Azul escuro	(0, 0, 128)
Azul petróleo	(0, 128, 128)
Azul-piscina	(0, 255, 255)
Branco	(0, 0, 0)
Cinza	(128, 128, 128)
Fúcsia	(255, 0, 255)
Marrom	(128, 0, 0)
Roxo	(128, 0, 128)
Verde	(0, 128, 0)
Verde claro (lime)	(0, 255, 0)
Verde oliva	(128, 128, 0)
Vermelho	(255, 0, 0)
Prateado	(192, 192, 192)
Preto	(0, 0, 0)

nome da fonte, que é utilizado `None`, para dizer ao programa que será utilizado o padrão do sistema. O segundo parâmetro o tamanho da fonte, dado como 48 pt.

10.8.2 O método `render()` para objetos `Font`

Na linha 22, o objeto `Font` armazenado em `basicFont`, chamou o método `render()`. Este método cria um objeto `Surface`, com o texto desejado. O primeiro parâmetro é a string a ser desenhada, o segundo é a escolha da aplicação da técnica *anti-aliasing*. O anti-aliasing é uma técnica que faz com que o desenho seja menos brusco, tornando as linhas mais suaves. Esta técnica é utilizada para o desenho da fonte. O terceiro e o quarto parâmetro são as cores para desenho do texto e do fundo do texto, respectivamente.

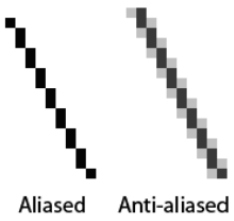

Figura 18: Técnica de *anti-aliasing* aplicada a uma reta.

10.9 Atributos

O tipo `pygame.Rect` (que será chamado de `Rect`, por conveniência), facilita o trabalho com formas retangulares. Para criar um objeto `Rect` chama-se a função `pygame.Rect()`, passando parâmetros inteiros de coordenadas XY do canto superior esquerdo, seguidos pela largura e altura do retângulo, da forma: `pygame.Rect(coordEsquerda, coordAcima, largura, altura)`.

Assim como métodos são como funções associadas a um objeto, **atributos** são variáveis associadas a um objeto. O tipo `rect` possui muitos atributos que descrevem um retângulo e são chamadas

como um ponto, assim como os métodos (ver linhas 24 e 25 do código 8). Observe a tabela 9, em que `myRect` é um objeto `Rect` qualquer.

Tabela 9: Atributos dos objetos `Rect`.

Atributo `pygame.Rect`	**Descrição**
myRect.left	O valor inteiro da coordenada X do lado esquerdo do retângulo
myRect.right	O valor inteiro da coordenada X do lado direito do retângulo
myRect.top	O valor inteiro da coordenada Y do lado de cima do retângulo
myRect.bottom	O valor inteiro da coordenada Y do lado de baixo do retângulo
myRect.centerx	O valor inteiro da coordenada X do centro do retângulo
myRect.centery	O valor inteiro da coordenada Y do lado centro do retângulo
myRect.width	O valor inteiro da largura do retângulo
myRect.height	O valor inteiro da altura do retângulo
myRect.size	Uma tupla de dois inteiros: `(largura, altura)`
myRect.topleft	Uma tupla de dois inteiros: `(esquerda, acima)`
myRect.topright	Uma tupla de dois inteiros: `(direita, acima)`
myRect.bottomleft	Uma tupla de dois inteiros: `(esquerda, abaixo)`
myRect.bottomright	Uma tupla de dois inteiros: `(direita, abaixo)`
myRect.midleft	Uma tupla de dois inteiros: `(esquerda, centroY)`
myRect.midright	Uma tupla de dois inteiros: `(direita, centroY)`
myRect.midtop	Uma tupla de dois inteiros: `(centroX, acima)`
myRect.midbottom	Uma tupla de dois inteiros: `(centroX, abaixo)`

10.10 Métodos `get_rect()` para os objetos `pygame.font.Font` e `pygame.Surface`

Observe que os objetos `Font` (armazenado na variável `text`) e `Surface` (armazenado na variável `windowSurface`) chamam o método `get_rect()`. Tecnicamente, eles são métodos diferentes. Entretanto, possuem o mesmo nome por executarem um procedimento similar, retornando objetos `Rect` que representam o tamanho e a posição de um objeto `Font` ou `Surface`.

Além disso, lembre-se que o `pygame` é um módulo importado pelo programa e, dentro deste módulo, estão disponíveis os módulos `font` e `surface`. E dentro destes módulos, estão os tipos `Font` e `Surface`. A diferença entre as letras maiúsculas no início dos objetos e minúsculas para os módulos é por convenção, para facilitar a distinção entre os tipos.

10.11 Função construtor e `type()`

Criou-se um objeto `pygame.Rect` através da chamada de função `pygame.Rect()`. A função `pygame.Rect()` possui o mesmo nome do "tipo de dado" `pygame.Rect`. Funções que possuem o mesmo nome de seu tipo de dado e criam objetos ou valores do seu tipo são chamadas de **construtores**.

As funções `int()` e `str()` também são construtores. A função `int()` retorna um inteiro do que estiver dentro dos parênteses, por exemplo, desde que especificado corretamente.

Para descobrir o tipo de um dado, utiliza-se a função `type()`:

```
1 >>> type('21')
2 <class 'str'>
3 >>> type(12)
4 <class 'int'>
5 >>> type(19.32)
6 <class 'float'>
```

```
7  >>> import pygame
8  >>> pygame.init()
9  (6, 0)
10 >>> retangulo = pygame.Rect(20, 20, 40, 40)
11 >>> type(retangulo)
12 <class 'pygame.Rect'>
13 >>> pygame.quit()
```

(É necessário chamar a função `pygame.quit()`) quando terminar de utilizar o módulo `pygame` no terminal interativo, para evitar que aconteça algum procedimento indesejado no Python.)

Observe que a função `type()` não retorna uma string, mas um valor do tipo `type`:

```
1 >>> type(type(12))
2 <class 'type'>
```

10.12 O método `fill()` para objetos `Surface`

A linha 28 traz o primeiro desenho do programa. O que esta linha faz é preencher a superfície inteira, armazenada em `windowSurface` com a cor branca (`WHITE`). O método `fill()` irá preencher completamente a superfície com a cor passada por parâmetro.

Algo importante a saber sobre o Pygame é que a janela na tela não mudará quando o método `fill()` for chamado, assim como qualquer outra função de desenho. Estes desenhos estão em um objeto `Surface`, mas este objeto não será desenhado na tela do usuário até que `pygame.display.update()` for chamado. Isto porque desenhar em um objeto `Surface` é muito mais rápido do que desenhar na tela do computador. Assim, é mais eficiente desenhar na tela "tudo de uma vez" depois que todos os desenhos já estejam em um `Surface`.

10.13 Mais funções

10.13.1 `pygame.draw.poligon()`

Um polígono é uma forma geométrica que possui vários lados e apenas linhas retas, em duas dimensões. A função `pygame.draw.poligon()` desenha qualquer forma e preenche o espaço dentro desta forma. A tupla de tuplas passada a esta função representa as coordanadas XY dos pontos da forma, em ordem. A última tupla conectará uma linha reta à primeira, formando um polígono. Observe a linha 31 do código 8.

10.13.2 Desenhos genéricos

Algumas funções de desenho são bem similares, exceto pela forma com que resultam e nos dados que precisam para gerar esta forma. Vejamos algumas delas:

- `pygame.draw.circle()`: desenha um círculo em um objeto `Surface` dado (linha 39). Parâmetros:
  1. Objeto `Surface` a ser desenhado
  2. Cor
  3. Tupla com as coordenadas inteiras X e Y indicando o centro do círculo
  4. Raio, em pixels
  5. Preenchimento, 0 (zero) significa que a forma será preenchida
- `pygame.draw.ellipse()`: desenha uma elipse em um objeto `Surface` dado (linha 42). Parâmetros:
  1. Objeto `Surface` a ser desenhado
  2. Cor
  3. Tupla com as coordenadas inteiras indicando limite superior, limite inferior, largura e altura
  4. Preenchimento, 1 (zero) significa que a forma não será preenchida

- `pygame.draw.rect()`: desenha um retângulo em um objeto `Surface` dado (linha 45). Parâmetros:
  1. Objeto `Surface` a ser desenhado
  2. Cor
  3. Tupla com as coordenadas inteiras indicando limite à esquerda, limite superior, largura e altura. Um objeto `Rect` também pode ser passado neste parâmetro
  4. Preenchimento, 1 (zero) significa que a forma não será preenchida

10.14 O tipo de dado `pygame.PixelArray`

Na linha 48, um objeto `pygame.PixelArray` é criado (que será chamado de `PixelArray` por conveniência). Este objeto é uma lista de listas de tuplas de cores que representam o objeto `Surface`, passado por parâmetro. O objeto `windowSurface` foi passado para o construtor `PixelArray()` na linha 48. Em seguida, atribuiu-se `BLACK` a `pixArray[480][380]`, para ser um pixel preto. O Pygame irá alterar automaticamente o objeto `windowSurface` para esta configuração.

O primeiro índice do objeto `PixelArray` é a coordenada X, o segundo, a coordenada Y. Objetos deste tipo tornam fáceis de gerar pixels individuais de um objeto em uma cor específica.

Criar um objeto `PixelArray` de um objeto `Surface` trava este segundo objeto. Isto significa que nenhuma chamada de função `blit()` (explicada adiante) pode ser feita ao objeto `Surface`. Para destravar o objeto, é necessário deletar o objeto `PixelArray` com o operador `del` (linha 50). Se isto não for feito, ocorrerá erro.

10.14.1 O método `blit()` para objetos `Surface`

O método `blit()` desenha conteúdo de um objeto `Surface` em outro objeto `Surface`. Por exemplo, na linha 54 é desenhado o texto `"Hello World!"`, armazenado na variável `text`, em um objeto `Surface` e, por sua vez, este objeto é armazenado na variável `windowSurface`.

O objeto `text` havia `"Hello World!"` desenhado na linha 22, pelo método `render()`. Objetos `Surface` são armazenados na memória (como qualquer outra variável), e não são desenhados na tela imediatamente. O objeto armazenado em `windowSurface` é desenhado na tela somente quando ocorre a chamada `pygame.display.update()`, na linha 56, pois este objeto foi criado pela função `pygame.display.set_mode()`.

O segundo parâmetro para `blit()` especifica onde, na superfície `windowSurface` (isto é, onde no objeto `Surface`), `text` deve ser desenhado. No caso da linha 53, o objeto `Rect` foi passado, obtido através da linha 23 em `text.get_Rect()`.

10.15 A função `pygame.display.update()`

No Pygame, nada é desenhado até que a função `pygame.display.update()` é chamada. Este procedimento é necessário pois desenhar na tela é uma operação muito lenta para o computador, comparada, por exemplo, à simples criação e armazenamento de objetos `Surface` na memória.

Para cada atualização da tela esta função deve ser chamada para mostrar os objetos `Surface` retornados por `pygame.display.set_mode()`. (Neste programa, há apenas o objeto armazenado em `windowSurface`.) Isto se tornará mais importante quando lidarmos com animações.

10.16 Eventos e loop do jogo

Nos programas anteriores, toda a saída era impressa até que houvesse uma chamada de `input()`. Neste ponto, o programa para e espera a entrada do usuário. O Pygame não funciona desta forma. Ao invés disso, os programas estão constantemente rodando em um loop (neste programa, executa-se todas as linhas de código cerca de cem vezes por segundo).

O **loop do jogo** é um laço que verifica constantemente por novos eventos, atualizações de estado da janela e desenha, o que for programado, na tela. **Eventos** são valores do tipo `pygame.event.Event`, gerados pelo Pygame sempre que algo ocorrer através da interface de usuário, como uma tecla pressi-

onada ou cliques e movimentos do mouse. Chamar `pygame.event.get()` retorna quaisquer novos objetos `pygame.event.Event` gerados desde a última chamada da mesma função.

O início deste programa é na linha 59, em um loop infinito. O programa terminará somente se um evento ocorrer, conforme declarado no bloco condicional seguinte.

10.16.1 A função `pygame.event.get()`

A função `pygame.event.get()` retorna uma lista de objetos `pygame.event.Event`. Esta lista possui todo evento ocorrido entre uma e outra chamada desta função. Todos os objetos `Event` possuem um atributo chamado `type`, que informa qual é o tipo de tal evento. Estes tipos serão vistos posteriormente, nesta aula veremos apenas o evento `QUIT`.)

O Pygame possui suas próprias constantes no módulo `pygame.locals`, que é importado na segunda linha do programa. A linha `from pygame.locals import *` significa que não é necessário digitar `pygame.locals` antes de utilizar qualquer recurso deste módulo. Por isso, utiliza-se apenas `QUIT` no programa.

Na linha 60, um loop `for` inicia e verifica cada objeto `Event` na lista retornada por `pygame.event.get()`. Se o atributo `type`[22] for igual ao conteúdo de `QUIT` (que é fornecido pelo módulo `pygame.locals`), então o programa saberá que o usuário quer fechar o programa.

O Pygame gera o evento `QUIT` quando o usuário clica no botão de fechar (X) no topo da janela do programa. Este evento também é gerado quando o computador está desligando e tenta terminar todos os programas que estão abertos. Por qualquer motivo que `QUIT` for gerado, sabe-se que o usuário deseja terminar/fechar o programa. Entretanto, pode-se simplesmente ignorar esta ação, o que causaria confusão ao usuário, provavelmente.

10.16.2 A função `pygame.quit()`

Se o evento `QUIT` for gerado, sabe-se que o usuário tentou fechar a janela. Neste caso, deve-se chamar a função do Pygame `pygame.quit()` e a função do Python `sys.exit()`.

Este foi o exemplo "`Hello World`" feito no Pygame. De fato, este programa não executa nada de útil, mas serviu para introduzir ferramentas gráficas muito úteis, com a finalidade de implementar recursos adicionais aos seus projetos. Embora possam parecer complicados, o Pygame fornece procedimentos que podem tornar os programas mais divertidos e interessantes do que os jogos anteriores, em modo texto.

Na próxima aula, veremos como lidar com animações.

10.17 Exercícios Complementares

1. Faça um programa que desenhe a seguinte situação de um jogo da velha na tela:

```
O|  |X
-----
O|X|
-----
O|X|O
```

   - O fundo da tela deve ser Branco
   - Deverá estar escrito centralizado em cima do tabuleiro o título "Jogo da Velha"na cor Preto.
   - O tabuleiro, no tamanho e na posição que você preferir, deverá estar na cor Steel Azul
   - As jogadas "X"deverão ser desenhados como duas linhas cruzadas e estar na cor Verde Hunter
   - As jogadas "O"deverão ser desenhadas círculos não preenchidos estar na cor Vermelho Indiano
   - Deverá ter um traço em cima do trio vencedor com uma cor aleatória e largura aleatória entre 1 e 10.

[22] Lembre-se que atributo é apenas uma variável que caracteriza um objeto.

2. Faça um programa que receba uma string do usuário e imprima cada letra em um local aleatório na tela. As as cores das letras devem ser aleatórias seguindo o seguinte padrão:
   Considere que há duas linhas verticais imaginarias na tela que a divide em três partes iguais:
   - Todas as letras que serão posicionadas antes da primeira linha devem ter o componente R da escala RGB maior que 127 e os outros menores
   - Todas as letras que serão posicionadas entre a primeira e a segunda linha devem ter o componente G maior que 127 e os outros menores
   - Todas as letras que serão posicionadas depois da segunda linha devem ter o componente B maior que 127 e os outros menores

11 Introdução à Gráficos e Animação (parte 2)

O programa deste capítulo é bem simples, em questão de execução. Alguns blocos "caminham" na tela, batendo nas bordas da janela. Os blocos são de diferentes tamanhos e cores e se movem em direções na diagonal. Para animar os blocos (isto é, dar movimentos a eles) será utilizado um loop, fazendo iterações nos pixels de cada forma. Desta forma, novos blocos serão desenhados a cada momento, alguns pontos à frente em relação à direção em que cada um estiver se movendo, dando a impressão de movimento.

11.1 Código-fonte

Código 9: Animações.

```
1  import pygame, sys, time
2  from pygame.locals import *
3
4  # set up pygame
5  pygame.init()
6
7  # set up the window
8  WINDOWWIDTH = 400
9  WINDOWHEIGHT = 400
10 windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT), 0, 32)
11 pygame.display.set_caption('Animation')
12
13 # set up direction variables
14 DOWNLEFT = 1
15 DOWNRIGHT = 3
16 UPLEFT = 7
17 UPRIGHT = 9
18
19 MOVESPEED = 4
20
21 # set up the colors
22 BLACK = (0, 0, 0)
23 RED = (255, 0, 0)
24 GREEN = (0, 255, 0)
25 BLUE = (0, 0, 255)
26
27 # set up the block data structure
28 b1 = {'rect':pygame.Rect(300, 80, 50, 100), 'color':RED, 'dir':UPRIGHT}
29 b2 = {'rect':pygame.Rect(200, 200, 20, 20), 'color':GREEN, 'dir':UPLEFT}
30 b3 = {'rect':pygame.Rect(100, 150, 60, 60), 'color':BLUE, 'dir':DOWNLEFT}
31 blocks = [b1, b2, b3]
32
33 # run the game loop
34 while True:
35     # check for the QUIT event
36     for event in pygame.event.get():
37         if event.type == QUIT:
38             pygame.quit()
39             sys.exit()
40
41     # draw the black background onto the surface
42     windowSurface.fill(BLACK)
43
44     for b in blocks:
```

```
45          # move the block data structure
46          if b['dir'] == DOWNLEFT:
47              b['rect'].left -= MOVESPEED
48              b['rect'].top += MOVESPEED
49          if b['dir'] == DOWNRIGHT:
50              b['rect'].left += MOVESPEED
51              b['rect'].top += MOVESPEED
52          if b['dir'] == UPLEFT:
53              b['rect'].left -= MOVESPEED
54              b['rect'].top -= MOVESPEED
55          if b['dir'] == UPRIGHT:
56              b['rect'].left += MOVESPEED
57              b['rect'].top -= MOVESPEED
58
59          # check if the block has move out of the window
60          if b['rect'].top < 0:
61              # block has moved past the top
62              if b['dir'] == UPLEFT:
63                  b['dir'] = DOWNLEFT
64              if b['dir'] == UPRIGHT:
65                  b['dir'] = DOWNRIGHT
66          if b['rect'].bottom > WINDOWHEIGHT:
67              # block has moved past the bottom
68              if b['dir'] == DOWNLEFT:
69                  b['dir'] = UPLEFT
70              if b['dir'] == DOWNRIGHT:
71                  b['dir'] = UPRIGHT
72          if b['rect'].left < 0:
73              # block has moved past the left side
74              if b['dir'] == DOWNLEFT:
75                  b['dir'] = DOWNRIGHT
76              if b['dir'] == UPLEFT:
77                  b['dir'] = UPRIGHT
78          if b['rect'].right > WINDOWWIDTH:
79              # block has moved past the right side
80              if b['dir'] == DOWNRIGHT:
81                  b['dir'] = DOWNLEFT
82              if b['dir'] == UPRIGHT:
83                  b['dir'] = UPLEFT
84
85          # draw the block onto the surface
86          pygame.draw.rect(windowSurface, b['color'], b['rect'])
87
88      # draw the window onto the screen
89      pygame.display.update()
90      time.sleep(0.02)
```

A execução deste programa resultará em algo parecido (e animado) com a figura 19.

11.2 Como o programa funciona

11.2.1 Movendo e rebatendo os blocos

Para gerar os movimentos dos blocos, é necessário que saibamos exatamente qual trajetória eles devem percorrer, além de ter uma boa noção espacial, para que a trajetória não siga de forma errada. Cada bloco, neste programa, se move em um de quatro movimentos na diagonal: abaixo e à esquerda, abaixo e à direita, acima e à esquerda, acima e à direita. Quando o bloco bate em algum lado da janela, ele deverá ser rebatido, mantendo-se dentro da janela e se movendo em uma nova direção na diagonal.

A nova direção do bloco depende de duas condições: qual é a direção que ele estava se movendo antes de ser rebatido e qual parede ele colidiu. Há um total de oito formas possíveis que um bloco

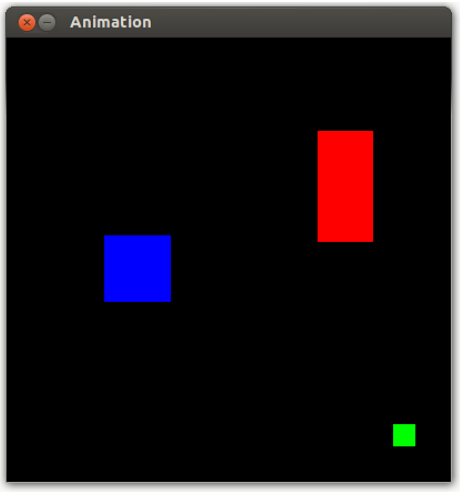

Figura 19: Resultado da execução do código 9.

pode colidir com uma parede: duas para cada uma das quatro paredes. Observe a figura 20. Por exemplo, se um bloco está se movendo para baixo e para a direita e colide com a parede debaixo da janela, então este bloco deverá ser rebatido para a direita e para cima.

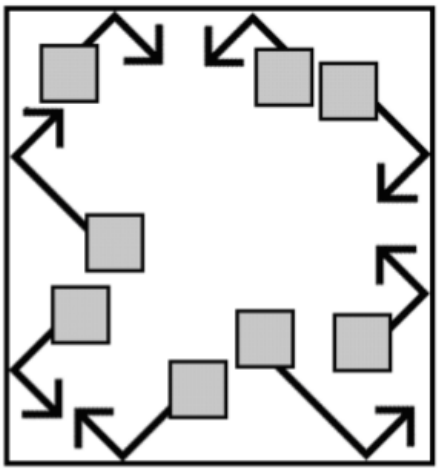

Figura 20: Formas de colisão possíveis neste programa.

Para representar os blocos, utiliza-se objetos `Rect`, para indicar as posições e tamanho de um bloco. Uma tupla de três inteiros representam a cor de um bloco e um inteiro representa qual das quatro direções o bloco está se movendo. A cada iteração do loop do jogo, as coordenadas X e Y dos blocos são ajustadas no objeto `Rect`. Além disso, a cada iteração os blocos são desenhados na nova posição. Com a execução contínua do programa, os blocos se movem gradualmente pela tela, parecendo que eles estão se movendo e sendo rebatidos ao colidirem com as paredes.

11.3 Como o programa funciona

11.3.1 Criando a janela principal e iniciando o Pygame

Neste programa, o módulo `time` é importado, além dos módulos do programa anterior.

Entre as linhas 8 e 10, determina-se o tamanho da janela, através das dimensões de altura e largura em `WINDOWWIDTH` e `WINDOWHEIGHT`, desenhando a janela em uma variável `windowSurface`. As dimensões são armazenadas em constantes pois estes valores serão utilizados além da função `set_mode()`. Desta forma, por conveniência, utilizam-se variáveis e constantes de forma que o

programa torna-se mais legível para o programador.

Se estas constantes não fossem utilizadas, por exemplo, cada ocorrência deveria deveria ser substituída pelo valor que elas carregam, o que não torna muito agradável para o programador, ainda mais no caso de um programa com uma grande densidade de linhas de código.

Para este programa, dá-se título de "Animation" para a janela, através da função `set_caption()`.

Em seguida, das linhas 15 a 18, constantes com números que simbolizam a posição da janela através da disposição dos números no teclado numérico (como no jogo da velha) são definidas para determinar a direção das formas. Por exemplo, `1` corresponde à direção abaixo e à esquerda, `3` abaixo e a direita, e assim por diante. Poderiam ser utilizados quaisquer valores para definir estas direções, desde que sejam valores diferentes. Eles servem para que o programa saiba em que direção seguir após uma colisão.

Uma outra constante, `MOVESPEED` é definida como a velocidade em que os blocos irão se mover. No caso, isto indica que os blocos se moverão 4 pixels a cada iteração do loop que faz o movimento dos blocos.

11.3.2 Iniciando as estruturas de dados dos blocos

A linha 28 inicia um dado do tipo dicionário (este tipo de dado já foi indicado como pesquisa, anteriormente) que representa um dos retângulos do programa. O mesmo é feito em seguida para os outros retângulos. O dicionário possuirá as chaves `rect` (com um objeto `Rect` atribuído) e `color` (com uma tupla de inteiros atribuída), além da direção inicial em `dir`.

Na linha 31, estes dados são armazenados em uma lista, `blocks`. Cada posição de `blocks` conterá uma variável do tipo dicionário. Ao realizar uma chamada do tipo `blocks[0]['color']`, o retorno seria `(255,0,0)`. Desta forma, quaisquer dos valores de qualquer bloco pode ser acessado, sendo iniciado por `blocks`.

11.3.3 Rodando o loop principal

Dentro do loop principal, queremos mover todos os blocos pela tela na direção em que eles foram definidos e, então, serem rebatidos assim que colidirem com uma das paredes. Assim, os blocos são desenhados na superfície e então a função `pygame.display.update()` para atualizar a imagem na tela. Além disso, a função `pygame.event.get()` para verificar se o usuário quer fechar o programa.

O loop `for` verifica todos os eventos da lista retornada por `pygame.event.get()`. Depois de ter desenhado os blocos em `windowSurface`, a base preta e, em seguida, redesenhar os retângulos nas novas posições.

11.3.4 Movendo cada bloco

O bloco `for` iniciado na linha 44 atualiza a posição de cada bloco. Deve-se percorrer a lista de retângulos e rodar o mesmo código em cada estrutura. Dentro do loop, cada bloco é representado por `b`, tornando mais fácil de digitar.

O novo valor de `left` e `top`, por exemplo, de cada retângulo (ou seja, os atributos), dependem da direção em que o bloco está se movendo. Lembre-se que as coordenadas X e Y iniciam em zero no extremo esquerdo superior da janela. Desta forma, se a direção do bloco for `DOWNLEFT` ou `DOWNRIGHT`, o atributo `top` será incrementado. Se a direção for `UPLEFT` ou `UPRIGHT`, queremos decrementar o atributo `top`.

Se a direção do bloco é `DOWNRIGHT` ou `UPRIGHT`, o atributo `left` será incrementado. Da mesma forma, se a direção for `DOWNLEFT` ou `UPLEFT`, o atributo `left` é decrementado.

Também seria possível modificar os atributos `right` e `bottom` ao invés de `top` e `left`, pois o Pygame irá atualizar o objeto `Rect` de qualquer forma.

11.3.5 Verificando se o bloco foi rebatido

O bloco `if` da linha 60 verifica se o bloco passou dos limites da janela. Além disso, dependendo da direção e da parede que o bloco colidir, uma nova direção é estabelecida de forma que dê a impressão de o bloco ter sido rebatido pela parede em que colidiu.

11.3.6 Mudando a direção do bloco rebatido

Observar a figura 20 facilita a visão da trajetória que os blocos devem seguir ao colidir com cada parede. Desta forma, a sequência de blocos `if` aninhados em cada bloco após a linha 60 estabelece as novas direções de cada bloco, dependendo da parede em que ocorrer a colisão e da direção em que eles chegam até ela.

11.3.7 Desenhando os blocos em suas novas posições

Depois de ter suas direções atualizadas, é necessário desenhar o bloco na tela. Lembre-se que o bloco é desenhado, primeiramente, em um objeto `Surface` para depois aparecer na tela para o usuário. Esta sequência de passos ocorre nas linhas 86 (fim do loop `for`) e 89 (que atualiza a superfície).

Finalmente, um intervalo de `0.02` segundos é dado para que a execução não ocorra muito rapidamente e seja inviável de observar o movimento dos blocos na tela. Tente comentar esta linha e ver como ocorre o movimento dos blocos.

11.4 Exercícios complementares

1. **(Vale 10% da nota e pode ser entregue em até duas semanas, a partir desta aula)** Modifique o programa da animação dos blocos de acordo com os seguintes itens:
   - Adicionar mais formas geométricas (pelo menos uma)
   - Formas geométricas em, pelo menos, 3 formas primitivas distintas
   - Velocidades distintas para cada forma
   - Condição de aparecer um texto caso uma situação específica ocorra (a sua escolha)
   - Condição de mudança de cor caso a forma colida com determinada parede (por exemplo, se a forma A colidir com a parede esquerda, ela deve se tornar azul), podendo ser apenas com uma parede específica, mas todas as formas devem sofrer esta alteração
   - (Associado com a condição anterior) Mudar a cor de fundo se a cor de uma, ou mais, formas também mudar (pode ser com uma forma específica – por exemplo, apenas se o triângulo bater em uma parede e ele mudar de cor, então a cor de fundo também mudará – ou mais de uma forma, senão todas), a sua escolha
   - **ADICIONAL (vale 5% a mais na nota e o aluno deve sinalizar que este item foi feito, na entrega do exercício):** as formas colidem umas com as outras e ambas tomam direções diferentes

12 Detecção de colisões

Tópicos abordados neste capítulo:

- Detecção de colisão
- Entrada por teclado
- Entrada por mouse

Um comportamento muito comum em jogos com recursos gráficos é a detecção de colisão, ou seja, quando dois objetos na tela se tocam. Este procedimento é muito comum no decorrer dos jogos. Por exemplo, se um jogador toca um inimigo, ele pode perder uma "vida" no jogo. Detectar colisão pode ajudar a determinar se o personagem está no chão ou se não há nada abaixo dele. O próximo programa cobre este fenômeno.

Posteriormente, serão detectadas as entradas do usuário através do teclado e mouse, que é mais complicado do que o uso da função `input()`.

12.1 Código fonte: colisão

Muito deste código é similar ao programa introdutório de animação, logo, o que diz respeito aos fenômenos relacionados à colisão nas paredes da janela, não será abordado novamente. Para iniciar, o programa será baseado no princípio do jogo *pacman*. Uma lista de objetos `pygame.Rect` representará as comidas desenhadas na tela. A cada iteração dentro do loop principal, o programa lerá cada objeto nesta lista e desenhar um retângulo verde na janela. A cada quarenta iterações do loop do jogo, o programa desenhará um novo quadrado verde na janela, assim, a janela sempre terá comidas.

Quem "comerá" os quadrados verdes (uma representação do pacman) será um quadrado branco, representado por um tipo `dictionary`, similar aos retângulos declarados no programa que tratava sobre animações, possuindo suas características e uma direção a ser seguida. Assim que o quadrado percorre a janela, verifica-se se ele colidiu com quaisquer dos quadrados que representam as comidas. Se isto ocorrer, o quadrado da comida é deletado.

O código que executa este procedimento está no código 10.

Código 10: Detecção de colisões.

```
1  import pygame, sys, random
2  from pygame.locals import *
3
4  def doRectsOverlap(rect1, rect2):
5      for a, b in [(rect1, rect2), (rect2, rect1)]:
6          # Check if a's corners are inside b
7          if ((isPointInsideRect(a.left, a.top, b)) or
8              (isPointInsideRect(a.left, a.bottom, b)) or
9              (isPointInsideRect(a.right, a.top, b)) or
10             (isPointInsideRect(a.right, a.bottom, b))):
11             return True
12
13     return False
14
15 def isPointInsideRect(x, y, rect):
16     if (x > rect.left) and (x < rect.right) and (y > rect.top) and (y < rect.
           bottom):
17         return True
18     else:
19         return False
20
21
22 # set up pygame
```

```
23 pygame.init()
24 mainClock = pygame.time.Clock()
25
26 # set up the window
27 WINDOWWIDTH = 400
28 WINDOWHEIGHT = 400
29 windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT), 0, 32)
30 pygame.display.set_caption('Collision Detection')
31
32 # set up direction variables
33 DOWNLEFT = 1
34 DOWNRIGHT = 3
35 UPLEFT = 7
36 UPRIGHT = 9
37
38 MOVESPEED = 4
39
40 # set up the colors
41 BLACK = (0, 0, 0)
42 GREEN = (0, 255, 0)
43 WHITE = (255, 255, 255)
44
45 # set up the bouncer and food data structures
46 foodCounter = 0
47 NEWFOOD = 40
48 FOODSIZE = 20
49 bouncer = {'rect':pygame.Rect(300, 100, 50, 50), 'dir':UPLEFT}
50 foods = []
51 for i in range(20):
52     foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH - FOODSIZE),
       random.randint(0, WINDOWHEIGHT - FOODSIZE), FOODSIZE, FOODSIZE))
53
54 # run the game loop
55 while True:
56     # check for the QUIT event
57     for event in pygame.event.get():
58         if event.type == QUIT:
59             pygame.quit()
60             sys.exit()
61
62     foodCounter += 1
63     if foodCounter >= NEWFOOD:
64         # add new food
65         foodCounter = 0
66         foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH - FOODSIZE),
           random.randint(0, WINDOWHEIGHT - FOODSIZE), FOODSIZE, FOODSIZE))
67
68     # draw the black background onto the surface
69     windowSurface.fill(BLACK)
70
71     # move the bouncer data structure
72     if bouncer['dir'] == DOWNLEFT:
73         bouncer['rect'].left -= MOVESPEED
74         bouncer['rect'].top += MOVESPEED
75     if bouncer['dir'] == DOWNRIGHT:
76         bouncer['rect'].left += MOVESPEED
77         bouncer['rect'].top += MOVESPEED
78     if bouncer['dir'] == UPLEFT:
79         bouncer['rect'].left -= MOVESPEED
80         bouncer['rect'].top -= MOVESPEED
81     if bouncer['dir'] == UPRIGHT:
82         bouncer['rect'].left += MOVESPEED
```

```
83          bouncer['rect'].top -= MOVESPEED
84
85      # check if the bouncer has move out of the window
86      if bouncer['rect'].top < 0:
87          # bouncer has moved past the top
88          if bouncer['dir'] == UPLEFT:
89              bouncer['dir'] = DOWNLEFT
90          if bouncer['dir'] == UPRIGHT:
91              bouncer['dir'] = DOWNRIGHT
92      if bouncer['rect'].bottom > WINDOWHEIGHT:
93          # bouncer has moved past the bottom
94          if bouncer['dir'] == DOWNLEFT:
95              bouncer['dir'] = UPLEFT
96          if bouncer['dir'] == DOWNRIGHT:
97              bouncer['dir'] = UPRIGHT
98      if bouncer['rect'].left < 0:
99          # bouncer has moved past the left side
100         if bouncer['dir'] == DOWNLEFT:
101             bouncer['dir'] = DOWNRIGHT
102         if bouncer['dir'] == UPLEFT:
103             bouncer['dir'] = UPRIGHT
104     if bouncer['rect'].right > WINDOWWIDTH:
105         # bouncer has moved past the right side
106         if bouncer['dir'] == DOWNRIGHT:
107             bouncer['dir'] = DOWNLEFT
108         if bouncer['dir'] == UPRIGHT:
109             bouncer['dir'] = UPLEFT
110
111     # draw the bouncer onto the surface
112     pygame.draw.rect(windowSurface, WHITE, bouncer['rect'])
113
114     # check if the bouncer has intersected with any food squares.
115     for food in foods[:]:
116         if doRectsOverlap(bouncer['rect'], food):
117             foods.remove(food)
118
119     # draw the food
120     for i in range(len(foods)):
121         pygame.draw.rect(windowSurface, GREEN, foods[i])
122
123     # draw the window onto the screen
124     pygame.display.update()
125     mainClock.tick(40)
```

A execução do código 10 gerará uma janela parecida com a figura 21.

12.2 Como o programa funciona

Este programa importa os mesmos módulos do programa de introdução à animação, incluindo o módulo `random`.

12.2.1 A função de detecção de colisão

Na linha 4, inicia-se uma função que determina se dois retângulos se interceptam, a `doRectsOverlap()`. A esta função são passados dois objetos `Rect` e ela retornará `True` se eles se interceptarem e `False`, se isto não ocorrer.

Há uma regra muito simples que determina se dois retângulos colidem: baseando-se nos quatro cantos de ambos, se um destes cantos estiver um dentro do outro retângulo, então eles colidiram. A partir desta análise, a resposta da função será `True` ou `False`. Esta verficação é realizada no bloco `if` iniciado na linha 7.

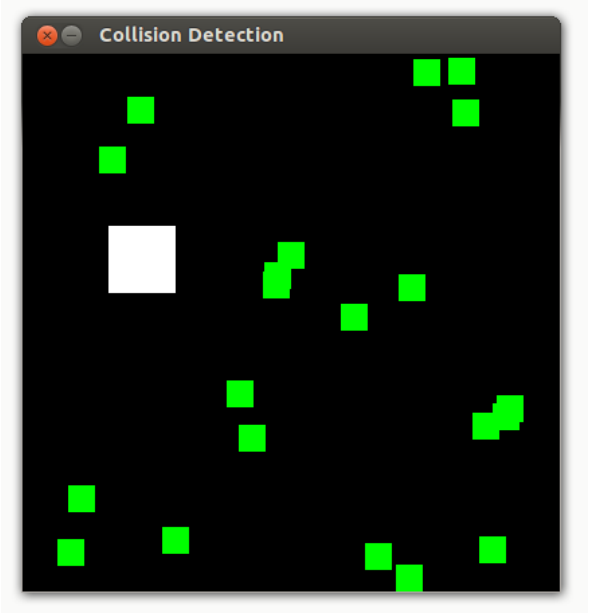

Figura 21: Resultado da execução do código 10.

12.2.2 Determinando se um ponto está dentro de um retângulo

A função `isPointInsideRect()` é utilizada pela função citada anteriormente, `doRectOverlap()`. Esta função retornará `True` se as coordenadas XY passadas pelos primeiro e segundo parâmetros estão dentro do objeto `pygame.Rect` passado como terceiro parâmetro, ou `False`, caso contrário.

Esta verificação é feita através do bloco `if` da função `isPointInsideRect()`. Se todas as condições forem verdadeiras, então o retorno será `True`, pois são condições interligadas com `and`. Se qualquer uma das condições for falsa, a sentença inteira também será.

Estas funções associadas (`doRectsOverlap()` e `isPointInsideRect()`) detectam a colisão entre dois retângulos e são utilizadas para quaisquer dois retângulos.

12.2.3 O objeto `pygame.time.Clock` e o método `tick()`

A maior parte do código entre as linhas 22 e 43 já foi visto no código sobre animação, da aula anterior. Entretanto, há um novo recurso na linha 24, a criação de um objeto `pygame.time.Clock`.

Anteriormente, havia sido utilizada a função `time.sleep()`, para diminuir a velocidade da execução e de forma que fosse possível que o usuário visualize o movimento das formas. Entretanto, o problema com esta função é que ela pode rodar muito rápido ou muito devagar, dependendo do desempenho do computador.

Para limitar o número máximo de iterações por segundo, utiliza-se o objeto `pygame.time.Clock`. A linha 125 chama `mainClock.tick(40)` dentro do loop do jogo. O método `tick()` verifica se o número de iterações por segundo foi o desejado (no caso, 40) durante o último segundo. Se o número de iterações for maior, então é colocado um pequeno intervalo entre as iterações, baseado no número de vezes que o método `tick()` foi chamado. Isto assegura que o programa nunca rode mais rápido do que o desejado.

12.2.4 Colidindo com os quadrados verdes

A linha 115 inicia o bloco que verifica se o quadrado branco colidiu com qualquer outro quadrado verde na tela. No caso de ter ocorrido colisão, o devido quadrado será eliminado da lista `foods`. Desta forma, o computador não desenhará os quadrados verdes que tenham sido "comidos".

12.2.5 Não adicione ou remova itens de uma lista enquanto houver iterações sobre ela

Observe que há algo um pouco diferente no loop da linha 115. Ao invés de fazer iterações sobre `foods`, ela é feita sobre zttfoods[:]. Basicamente, `foods[:]` cria uma nova lista com uma cópia dos itens de `foods`. Esta é uma forma resumida de fazer uma cópia de uma lista.

Mas qual é o motivo de não fazer iterações na lista original? Esta medida é tomada pois não é possível adicionar ou remover itens de uma lista enquanto há iterações sobre ela (ou seja, quando ela faz parte da declaração de um loop).

12.2.6 Removendo os quadrados verdes

A linha 116 é onde a função `doRectsOverlap()` é utilizada. Dois parâmetros são passados para a função, o quadrado branco e o quadrado verde mais próximo. Se estes dois retângulos colidirem, então o retorno será `True` e o quadrado verde será removido da lista.

12.2.7 Desenhando os quadrados verdes na tela

As linhas 120 e 121 são bem similares em relação a como se desenham os retângulos na tela. O vetor `foods` é percorrido e então um quadrado verde é desenhado na `windowSurface`.

Os últimos programas são interessantes para vê-los funcionando, entretanto, não há nenhuma interação com o usuário. Para introduzir estes conceitos, o próximo programa tratará de eventos obtidos através do teclado.

12.3 Código-fonte: entrada do usuário

A execução deste programa é similar ao anterior, com a diferença que o retângulo maior se movimenta a partir das setas que são recebidas como entrada do usuário. Uma alternativa para os movimentos são as letras `W, A, S, D`.

Código 11: Detecção de entrada do usuário.

```
1  import pygame, sys, random
2  from pygame.locals import *
3
4  # set up pygame
5  pygame.init()
6  mainClock = pygame.time.Clock()
7
8  # set up the window
9  WINDOWWIDTH = 400
10 WINDOWHEIGHT = 400
11 windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT), 0, 32)
12 pygame.display.set_caption('Input')
13
14 # set up the colors
15 BLACK = (0, 0, 0)
16 GREEN = (0, 255, 0)
17 WHITE = (255, 255, 255)
18
19 # set up the player and food data structure
20 foodCounter = 0
21 NEWFOOD = 40
22 FOODSIZE = 20
23 player = pygame.Rect(300, 100, 50, 50)
24 foods = []
25 for i in range(20):
26     foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH - FOODSIZE),
           random.randint(0, WINDOWHEIGHT - FOODSIZE), FOODSIZE, FOODSIZE))
27
```

```
28 # set up movement variables
29 moveLeft = False
30 moveRight = False
31 moveUp = False
32 moveDown = False
33
34 MOVESPEED = 6
35
36
37 # run the game loop
38 while True:
39     # check for events
40     for event in pygame.event.get():
41         if event.type == QUIT:
42             pygame.quit()
43             sys.exit()
44         if event.type == KEYDOWN:
45             # change the keyboard variables
46             if event.key == K_LEFT or event.key == ord('a'):
47                 moveRight = False
48                 moveLeft = True
49             if event.key == K_RIGHT or event.key == ord('d'):
50                 moveLeft = False
51                 moveRight = True
52             if event.key == K_UP or event.key == ord('w'):
53                 moveDown = False
54                 moveUp = True
55             if event.key == K_DOWN or event.key == ord('s'):
56                 moveUp = False
57                 moveDown = True
58         if event.type == KEYUP:
59             if event.key == K_ESCAPE:
60                 pygame.quit()
61                 sys.exit()
62             if event.key == K_LEFT or event.key == ord('a'):
63                 moveLeft = False
64             if event.key == K_RIGHT or event.key == ord('d'):
65                 moveRight = False
66             if event.key == K_UP or event.key == ord('w'):
67                 moveUp = False
68             if event.key == K_DOWN or event.key == ord('s'):
69                 moveDown = False
70             if event.key == ord('x'):
71                 player.top = random.randint(0, WINDOWHEIGHT - player.height)
72                 player.left = random.randint(0, WINDOWWIDTH - player.width)
73
74         if event.type == MOUSEBUTTONUP:
75             foods.append(pygame.Rect(event.pos[0], event.pos[1], FOODSIZE,
                FOODSIZE))
76
77     foodCounter += 1
78     if foodCounter >= NEWFOOD:
79         # add new food
80         foodCounter = 0
81         foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH - FOODSIZE),
               random.randint(0, WINDOWHEIGHT - FOODSIZE), FOODSIZE, FOODSIZE))
82
83     # draw the black background onto the surface
84     windowSurface.fill(BLACK)
85
86     # move the player
87     if moveDown and player.bottom < WINDOWHEIGHT:
```

```
88          player.top += MOVESPEED
89      if moveUp and player.top > 0:
90          player.top -= MOVESPEED
91      if moveLeft and player.left > 0:
92          player.left -= MOVESPEED
93      if moveRight and player.right < WINDOWWIDTH:
94          player.right += MOVESPEED
95
96      # draw the player onto the surface
97      pygame.draw.rect(windowSurface, WHITE, player)
98
99      # check if the player has intersected with any food squares.
100     for food in foods[:]:
101         if player.colliderect(food):
102             foods.remove(food)
103
104     # draw the food
105     for i in range(len(foods)):
106         pygame.draw.rect(windowSurface, GREEN, foods[i])
107
108     # draw the window onto the screen
109     pygame.display.update()
110     mainClock.tick(40)
```

12.3.1 Iniciando os movimentos

Entre as linhas 29 e 33, são definidos os movimentos iniciais do quadrado branco. Quatro variáveis diferentes são utilizadas para indicar as direções para cima, para baixo, para a esquerda e para a direita. Os movimentos (direções) iniciais são definidos como `False` de forma que, a cada momento que uma das teclas indicadas fora acionada, a variável que for relacionada a tal direção será dada como `True`.

As linhas 34 a 43 são idênticas aos códigos anteriores.

Tarefa 12.1
Pesquise detalhes sobre os eventos de mouse e teclado no Pygame (QUIT, KEYDOWN, KEYUP, MOUSEMOTION, MOUSEBUTTONDOWN e MOUSEBUTTONUP, principalmente) e causas que podem gerá-los. Este tópico pode ser útil para o trabalho final.

12.3.2 Lidando com as variáveis do teclado

Os blocos `if` iniciados na linha 44 até a linha 57 atribuem os movimentos às variáveis definidas no início do programa. Primeiro, é verificado se alguma tecla é pressionada, através do evento `KEYDOWN`.

Adicionalmente, o bloco `if` da linha 58 verifica se alguma tecla deixou de ser pressionada. Em caso afirmativo, ainda é verificado se a próxima tecla a ser pressionada é a tecla `Esc`, além dos movimentos comuns. Isto quer dizer que o usuário quer fechar o programa através do comando do teclado[23].

12.3.3 O evento `MOUSEBUTTONUP`

Eventos gerados pelo mouse são tratados da mesma forma que eventos de teclado. O evento `MOUSEBUTTONUP` ocorre quando o usuário clica sobre a janela do programa. O atributo `pos` no objeto `Event` é uma tupla de dois inteiros que carregam as coordenadas XY do clique. Na linha 75, observa-se que a coordenada X é armazenada em `event.pos[0]` e, a coordenada Y, em `event.pos[1]`. Uma novo objeto `Rect` é criado para representar um novo quadrado verde onde o evento ocorreu. Adicionando um novo objeto `Rect` à lista `foods`, um novo quadrado verde aparecerá na tela.

[23] Descubra o que acontece quando a tecla 'x' é pressionada e como isto acontece

12.3.4 Movendo o "jogador" pela tela

A sequência de blocos `if` entre as linhas 87 e 94 definem se o jogador está pressionando as setas ou não. O quadrado do jogador será movimentado a partir do quadrado armazenado em `player`, ajustando as coordenadas XY desta variável.

12.3.5 O método `colliderect()`

No código 10, fizemos nossa própria função para verificar se um retângulo intercepta outro (colide). Esta função foi incluída nesta aula para que você saiba como funciona o procedimento de analisar a colisão entre doia objetos na tela. Neste programa, será utilizada a função que já detecta colisão, do Pygame. O método `colliderect()` para objetos `pygame.Rect` recebe um segundo retângulo como parâmetro e verifica se eles colidem, utilizado no loop `for` que se inicia na linha 100. Ele retorna `True` se existe colisão. É o mesmo procedimento que foi feito na função `doRectOverlap()` com o auxílio da função `isPointInsideRect()`, do código 10.

O restante do código é similar ao código 10.

12.4 Exercícios complementares

1. Altere o código 11 de acordo com as seguintes situações:
   - Cada quadrado verde capturado soma um ponto e esta pontuação é mostrada para o usuário na janela do jogo
   - Insira quadrados de diferentes cores (pode ser apenas mais uma cor), e a cor adicional possuirá pontuação diferente dos quadrados verdes (por exemplo, quadrado verde vale um ponto e quadrados vermelhos valem 2 pontos) e tempo para aparecer na tela diferente dos quadrados verdes.
   - Insira um quadrado de uma cor diferente das utilizadas, que se move aleatoriamente (você define este aleatório, que pode não ser tão aleatório assim) e, se o jogador colidir com este quadrado, ele perde pontos.

13 Dodger

Tópicos abordados neste capítulo:

- A flag `pygame.FULLSCREEN`
- Constantes do Pygame para o teclado
- O método `move_ip()` para objetos `Rect`
- A função `pygame.mouse.set_pos()`
- Implementação de "cheat codes" em seus jogos
- Modificar o jogo *Dodger*

Neste capítulo, muitos dos conceitos apresentados anteriormente serão aplicados para desenvolver um jogo mais sofisticado, utilizando grande parte dos recursos aprendidos nos capítulos que abordaram interface gráfica.

Este jogo possui como jogador um pequeno homem que deve se esquivar do monte de blocos que caem do topo da tela. Quanto maior o tempo que o jogador se mantém desviando dos blocos, maior a sua pontuação.

Alguns "cheats" também foram colocados no jogo. Experimente pressionar as teclas `x` ou `z`.

13.1 Revisão dos tipos básicos de dados do Pygame

A seguir, há um resumo de alguns tipos básicos de recursos do Pygame vistos até agora.

- `pygame.Rect`: objetos do tipo `Rect` representam um espaço retangular com localização e tamanho. A localização deste tipo de objeto pode ser obtida através dos atributos dos objetos `Rect` (como `topleft`, `topright`, `bottomleft` e `bottomright`). Estes atributos são uma tupla de inteiros com coordenadas XY. O tamanho pode ser determinado pelos atributos `width` e `height`, que retornam valores em pixels. Objetos `Rect` possuem o método `colliderect()`, que verificam se há intersecção entre dois destes objetos.
- `pygame.Surface`: objetos `Surface` são áreas de pixels coloridos. Estes objetos representam uma imagem retangular, enquanto um objeto `Rect` representa apenas um espaço e localização de um retângulo. Objetos `Surface` possuem o método `blit()` que é utilizado para desenhar uma imagem de um objeto `Surface` em outro objeto `Surface`.
- Lembre-se que os objetos `Surface` possuem "coisas" desenhadas que não são possíveis de ver enquanto estão apenas armazenadas na memória do computador. É necessário "mandar" que o computador as mostre para o usuário, através do método `blit()`.
- `pygame.event.Event`: os tipos de dados `Event`, do módulo `pygame.Event`, gera eventos a cada vez que há uma interação do usuário com o programa. A função `pygame.event.get()` retorna uma lista de objetos `Event`.
- `pygame.font.Font`: o módulo `pygame.font` possui o tipo de dado `Font` que representa um texto formatado no Pygame. Cria-se um objeto `Font` chamando o construtor `pygame.font.SysFont()`. Os argumentos passados são uma string e um inteiro, com o nome da fonte e o tamanho, respectivamente. É comum passar o valor `None` para o nome da fonte, para utilizar a fonte padrão do sistema.
- `pygame.time.Clock`: o objeto `Clock`, do módulo `pygame.time`, é útil para manter os jogos rodando o mais rápido possível. Entretanto, isto pode não ser um recurso muito desejável, visto que há o risco de o jogador não conseguir acompanhar o jogo adequadamente. Os objetos `Clock` possuem o método `tick()`, que controla quantos frames por segundo (fps) o programador quer que o jogo execute. Quanto maior o fps, maior a velocidade do jogo. Utilizou-se 40 fps nos jogos anteriores. Observe que o módulo `pygame.time` é diferente daquele módulo `time` que contém a função `sleep()`.

13.2 Código-fonte

Novamente, baixe os recursos de imagem e som para este programa no site http://inventwithpython.com/resources.

Código 12: Jogo "Dodger".

```
1  import pygame, random, sys
2  from pygame.locals import *
3
4  WINDOWWIDTH = 600
5  WINDOWHEIGHT = 600
6  TEXTCOLOR = (255, 255, 255)
7  BACKGROUNDCOLOR = (0, 0, 0)
8  FPS = 40
9  BADDIEMINSIZE = 10
10 BADDIEMAXSIZE = 40
11 BADDIEMINSPEED = 1
12 BADDIEMAXSPEED = 8
13 ADDNEWBADDIERATE = 6
14 PLAYERMOVERATE = 5
15
16 def terminate():
17     pygame.quit()
18     sys.exit()
19
20 def waitForPlayerToPressKey():
21     while True:
22         for event in pygame.event.get():
23             if event.type == QUIT:
24                 terminate()
25             if event.type == KEYDOWN:
26                 if event.key == K_ESCAPE: # pressing escape quits
27                     terminate()
28                 return
29
30 def playerHasHitBaddie(playerRect, baddies):
31     for b in baddies:
32         if playerRect.colliderect(b['rect']):
33             return True
34     return False
35
36 def drawText(text, font, surface, x, y):
37     textobj = font.render(text, 1, TEXTCOLOR)
38     textrect = textobj.get_rect()
39     textrect.topleft = (x, y)
40     surface.blit(textobj, textrect)
41
42 # set up pygame, the window, and the mouse cursor
43 pygame.init()
44 mainClock = pygame.time.Clock()
45 windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
46 pygame.display.set_caption('Dodger')
47 pygame.mouse.set_visible(False)
48
49 # set up fonts
50 font = pygame.font.SysFont(None, 48)
51
52 # set up sounds
53 gameOverSound = pygame.mixer.Sound('gameover.wav')
54 pygame.mixer.music.load('background.mid')
55
56 # set up images
```

```
57 playerImage = pygame.image.load('player.png')
58 playerRect = playerImage.get_rect()
59 baddieImage = pygame.image.load('baddie.png')
60
61 # show the "Start" screen
62 drawText('Dodger', font, windowSurface, (WINDOWWIDTH / 3), (WINDOWHEIGHT / 3)
       )
63 drawText('Press a key to start.', font, windowSurface, (WINDOWWIDTH / 3) -
       30, (WINDOWHEIGHT / 3) + 50)
64 pygame.display.update()
65 waitForPlayerToPressKey()
66
67
68 topScore = 0
69 while True:
70     # set up the start of the game
71     baddies = []
72     score = 0
73     playerRect.topleft = (WINDOWWIDTH / 2, WINDOWHEIGHT - 50)
74     moveLeft = moveRight = moveUp = moveDown = False
75     reverseCheat = slowCheat = False
76     baddieAddCounter = 0
77     pygame.mixer.music.play(-1, 0.0)
78
79     while True: # the game loop runs while the game part is playing
80         score += 1 # increase score
81
82         for event in pygame.event.get():
83             if event.type == QUIT:
84                 terminate()
85
86             if event.type == KEYDOWN:
87                 if event.key == ord('z'):
88                     reverseCheat = True
89                 if event.key == ord('x'):
90                     slowCheat = True
91                 if event.key == K_LEFT or event.key == ord('a'):
92                     moveRight = False
93                     moveLeft = True
94                 if event.key == K_RIGHT or event.key == ord('d'):
95                     moveLeft = False
96                     moveRight = True
97                 if event.key == K_UP or event.key == ord('w'):
98                     moveDown = False
99                     moveUp = True
100                if event.key == K_DOWN or event.key == ord('s'):
101                    moveUp = False
102                    moveDown = True
103
104            if event.type == KEYUP:
105                if event.key == ord('z'):
106                    reverseCheat = False
107                    score = 0
108                if event.key == ord('x'):
109                    slowCheat = False
110                    score = 0
111                if event.key == K_ESCAPE:
112                        terminate()
113
114                if event.key == K_LEFT or event.key == ord('a'):
115                    moveLeft = False
116                if event.key == K_RIGHT or event.key == ord('d'):
```

```
117                     moveRight = False
118                 if event.key == K_UP or event.key == ord('w'):
119                     moveUp = False
120                 if event.key == K_DOWN or event.key == ord('s'):
121                     moveDown = False
122
123             if event.type == MOUSEMOTION:
124                 # If the mouse moves, move the player where the cursor is.
125                 playerRect.move_ip(event.pos[0] - playerRect.centerx, event.
                        pos[1] - playerRect.centery)
126
127         # Add new baddies at the top of the screen, if needed.
128         if not reverseCheat and not slowCheat:
129             baddieAddCounter += 1
130         if baddieAddCounter == ADDNEWBADDIERATE:
131             baddieAddCounter = 0
132             baddieSize = random.randint(BADDIEMINSIZE, BADDIEMAXSIZE)
133             newBaddie = {'rect': pygame.Rect(random.randint(0, WINDOWWIDTH-
                    baddieSize), 0 - baddieSize, baddieSize, baddieSize),
134                     'speed': random.randint(BADDIEMINSPEED,
                            BADDIEMAXSPEED),
135                     'surface':pygame.transform.scale(baddieImage, (
                            baddieSize, baddieSize)),
136                     }
137
138             baddies.append(newBaddie)
139
140         # Move the player around.
141         if moveLeft and playerRect.left > 0:
142             playerRect.move_ip(-1 * PLAYERMOVERATE, 0)
143         if moveRight and playerRect.right < WINDOWWIDTH:
144             playerRect.move_ip(PLAYERMOVERATE, 0)
145         if moveUp and playerRect.top > 0:
146             playerRect.move_ip(0, -1 * PLAYERMOVERATE)
147         if moveDown and playerRect.bottom < WINDOWHEIGHT:
148             playerRect.move_ip(0, PLAYERMOVERATE)
149
150         # Move the mouse cursor to match the player.
151         pygame.mouse.set_pos(playerRect.centerx, playerRect.centery)
152
153         # Move the baddies down.
154         for b in baddies:
155             if not reverseCheat and not slowCheat:
156                 b['rect'].move_ip(0, b['speed'])
157             elif reverseCheat:
158                 b['rect'].move_ip(0, -5)
159             elif slowCheat:
160                 b['rect'].move_ip(0, 1)
161
162          # Delete baddies that have fallen past the bottom.
163         for b in baddies[:]:
164             if b['rect'].top > WINDOWHEIGHT:
165                 baddies.remove(b)
166
167         # Draw the game world on the window.
168         windowSurface.fill(BACKGROUNDCOLOR)
169
170         # Draw the score and top score.
171         drawText('Score: %s' % (score), font, windowSurface, 10, 0)
172         drawText('Top Score: %s' % (topScore), font, windowSurface, 10, 40)
173
174         # Draw the player's rectangle
```

```
175             windowSurface.blit(playerImage, playerRect)
176
177             # Draw each baddie
178             for b in baddies:
179                 windowSurface.blit(b['surface'], b['rect'])
180
181             pygame.display.update()
182
183             # Check if any of the baddies have hit the player.
184             if playerHasHitBaddie(playerRect, baddies):
185                 if score > topScore:
186                     topScore = score # set new top score
187                 break
188
189             mainClock.tick(FPS)
190
191         # Stop the game and show the "Game Over" screen.
192         pygame.mixer.music.stop()
193         gameOverSound.play()
194
195         drawText('GAME OVER', font, windowSurface, (WINDOWWIDTH / 3), (
                WINDOWHEIGHT / 3))
196         drawText('Press a key to play again.', font, windowSurface, (WINDOWWIDTH
                / 3) - 80, (WINDOWHEIGHT / 3) + 50)
197         pygame.display.update()
198         waitForPlayerToPressKey()
199
200         gameOverSound.stop()
```

A execução deste programa é parecida com a figura 22.

Figura 22: Execução do programa Dodger.

13.3 Como o programa funciona e recursos adicionais

Este programa aborda praticamente todos os conceitos já estudados, com alguns recursos adicionais. Observe como algumas técnicas são utilizadas a fim de facilitar a prática da programação e, consequentemente, diminuir a taxa de erros que podem ocorrer em um programa.

13.3.1 Importando os módulos

Este jogo importa todos os módulos que os jogos anteriores com o Pygame utilizavam: `pygame`, `random`, `sys` e `pygame.locals`. O módulo `pygame.locals` possui várias constantes que o Pygame utiliza como tipos de evento e tipos de teclas. Utilizando a sentença `from pygame.locals import *`, podemos utilizar a constante `QUIT` ao invés de `pygame.locals.QUIT`.

13.3.2 Explorando o uso de constantes

Há várias constantes neste jogo. Utilizam-se constantes em um programa para facilitar o manuseio de muitas variáveis. Estas ainda, que podem ter o mesmo valor, podem ser alteradas a desejo do programador e não é desejável percorrer todo o programa para procurar tais valores. Desta forma, utilizam-se constantes (ou variáveis) declaradas, geralmente, no início do programa ou de um bloco, e, ao invés de escrever sempre um mesmo valor, escreve-se a constante que o carrega. Assim, ao alterar uma constante, em todos os locais que ela for citada, no programa, receberão o novo valor.

Observe as linhas 4 a 14 do código 12.

13.3.3 Explorando o uso de funções

Funções, assim como as variáveis, auxiliam o programador a não precisar digitar um mesmo trecho de código mais de uma vez, evitando erros de sintaxe e/ou lógica no programa. Observe as funções `terminate()`, `waitForPlayerToPressKey()`, `playerHasHitBaddie()` e `drawText()`. As funções que o programador criar em seu programa devem ter nomes que sugerem o que será executado em cada trecho de código, a fim de aumentar a legibilidade do código. Todas as funções deste programa abordam tópicos que já foram vistos anteriormente.

13.3.4 Cursor do mouse

Neste programa, é possível controlar a personagem do jogador com o mouse. A seta do mouse não é mostrada. Entretanto, se for desejado que a seta apareça, basta mudar a linha 47 para

```
47 pygame.display.set_visible(True)
```

13.3.5 Modo fullscreen

A função `pygame.display.set_mode()` possui um segundo parâmetro, opcional. Altere a linha 45 do programa, da seguinte forma:

```
45 pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT), pygame.FULLSCREEN)
```

13.3.6 Mostrando a tela inicial

O código entre as linhas 62 e 65 chama as funções necessárias para mostrar a tela inicial, que aguarda o usuário pressionar alguma tela, para, então, iniciar o jogo.

Observe que, nas linhas 62 e 63, chamou-se a função `drawtext()`, passando cinco argumentos: a string para a aparecer na tela, o tipo de fonte, o objeto `Surface` para projetar o texto, e as coordenadas XY do objeto `Surface` para que o texto seja desenhado.

A função `waitForPlayerToPressKey()` verificará os eventos de pressão de tecla (`KEYDOWN`) no programa. Dependendo da interação do jogador, a execução ou termina o jogo ou prossegue no loop do jogo.

A tabela 10 mostra alguns dos principais eventos de teclado que podem ser úteis para os seus jogos.

Tabela 10: Constantes de evento para as teclas pressionadas.

Constante do Pygame	Tecla	Constante do Pygame	Tecla
K_LEFT	Seta esquerda	K_HOME	Home
K_RIGHT	Seta direita	K_END	End
K_UP	Seta para cima	K_PAGEUP	Page Up
K_DOWN	Seta para baixo	K_PAGEDOWN	Page Down
K_ESCAPE	Esc	K_F1	F1
K_BACKSPACE	Backspace	K_F2	F2
K_TAB	Tab	K_F3	F3
K_RETURN	Return ou Enter	K_F4	F4
K_SPACE	Barra de espaço	K_F5	F5
K_DELETE	Del	K_F6	F6
K_LSHIFT	Shift da esquerda	K_F7	F7
K_RSHIFT	Shift da direita	K_F8	F8
K_LCTRL	Ctrl da esquerda	K_F9	F9
K_RCTRL	Ctrl da direita	K_F10	F10
K_LALT	Alt da esquerda	K_F11	F11
K_RALT	Alt da direita	K_F12	F12

13.3.7 O método `move_ip()` para objetos `Rect` e evento de movimento do mouse

Os eventos de teclado já foram abordados anteriormente. Mas, e se o personagem também puder ser controlado pelo mouse? Isto é feito no bloco entre as linhas 123 e 125.

No jogo *Dodger*, não há efeitos se o mouse for clicado, mas ele responde aos movimentos do mouse. Isto dá ao jogador outra forma de controle no jogo, além do teclado.

O evento que controla este movimento é do tipo `MOUSEMOTION`. Um evento deste é gerado a cada vez que o mouse é movido. Objetos `Event` do tipo `MOUSEMOTION` possuem um atributo chamado `pos`. Este atributo armazena uma tupla de coordenadas XY da posição do cursor do mouse na tela.

O método `move_ip()`, para objetos `Rect`, moverá a localização de um objeto `Rect` horizontalmente ou verticalmente, por um número de pixels. Por exemplo, `playerRect.move_ip(10, 20)` moverá o objeto 10 pixels para a direita e 20 pixels abaixo. Para mover o objeto para cima ou para baixo, basta passar valores negativos. Por exemplo, `playerRect.move_ip(-5, -15)` moverá o objeto `Rect` 5 pixels para a esquerda e 15 pixels acima.

O *ip*, deste método, se refere à "in place". Isto é devido ao método mudar o próprio objeto de local. Existe, ainda, o método `move()`, que não muda um objeto `Rect`, mas cria outro com uma nova posição. Este método é útil quando se deseja preservar a localização de um objeto `Rect` e obter um novo objeto em um novo local.

13.3.8 Movendo a personagem do jogador

Os blocos entre as linhas 141 e 148 verificam os movimentos do jogador e determinam as margens em que ele pode se mover (limites da janela). O método `move_ip()` é utilizado para movimentar a personagem do jogador.

13.3.9 A função `pygame.mouse.set_pos()`

A linha 151 move o cursor do mouse de acordo com a posição da personagem do jogador. A função `pygame.mouse.set_pos()` move o mouse às coordenadas XY lhe passadas como argumento. Especificamente, o cursor estará no meio da imagem da personagem do jogador se forem passadas as posições `playerRect.centerx` e `playerRect.centery`. O cursos existe e pode ser

movido mesmo que a configuração `pygame.mouse.set_visible(False)` tenha sido estabelecida previamente.

O motivo de fazer o cursor do mouse acompanhar a localização do jogador é evitar "pulos". Imagine que o jogador inicie o jogo com o controle do teclado e, no meio do jogo, decida mudar para o mouse. Dependendo de onde o ponteiro do mouse estiver, a personagem do jogador poderá sofrer um salto para esta posição (pois, provavelmente, a posição do cursor não será a mesma que o jogador parou).

13.3.10 Desenhando o placar do jogo

As linhas 171 e 172 mandam as informações, para serem mostradas no placar, à função `drawText()`. Observe que esta função já foi utilizada algumas vezes e como seria inconveniente digitar sempre os mesmos comandos para mostrar um texto na tela. Novamente, observa-se a facilidade que as funções trazem a um programa.

13.4 Finalizando o projeto

Você já está apto a entender o restante das linhas de código do programa que não foram citadas nesta aula, visto que muitas delas se baseiam em conceitos já apresentados anteriormente. Exercite a sua lógica e faça o **teste de mesa** deste programa para realmente entendê-lo. Você perceberá que esta prática irá ajudar-lhe a aprimorar seu entendimento das práticas e da lógica de programação.

Aguardamos pela apresentação do seu projeto final, com as habilidades aprendidas ao longo das aulas e até com recursos não apresentados durante o percurso.

A seguir, alguns links que podem contribuir para o seu conhecimento acerca do assunto:

- http://gamedevlessons.com
- http://www.hobbygamedev.com/
- http://www.pygame.org/
- http://pyopengl.sf.net

O grupo PET-CoCE espera que você tenha feito bom proveito desta oficina!

A Fluxogramas

Conforme citado no 6, um fluxograma é uma maneira de representar um algoritmo[24] em forma gráfica. Cada caixa representa um passo, e cada seta, a sequência do algoritmo. É nesta fase do desenvolvimento que pensamos nas estratégias pra resolver o problema proposto.

A.1 Criando um Fluxograma

Tenha em mente que seus fluxogramas nem sempre devem estar exatamente como este. Você apenas deve compreender que o fluxograma que você fez será útil enquanto estiver programando.

Veja o seguinte exemplo da criação do fluxograma do jogo da forca implementado no 6:

1. Em todo fluxograma devemos representar as caixas de início e fim, como segue o exemplo na figura 23.

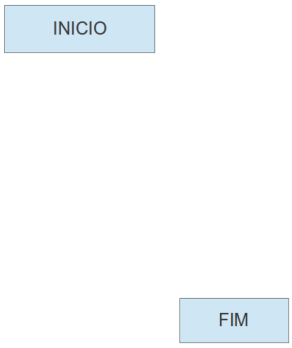

Figura 23: Começando seu fluxograma com as caixas início e fim.

2. Agora vamos pensar na estratégia algorítmica para resolver o problema (no caso programar o jogo da forca). Primeiro, um jogador pensa numa palavra secreta, desenha a forca e desenha os espaços correspondentes a cada letra (neste caso será o computador). Então o segundo jogador (o usuário do jogo) tentará adivinhar as letras. Também devemos adicionar as caixas com as ações para estes eventos. O fluxograma fica igual à figura 24.

[24] Um algoritmo é uma sequência de passos para a realização de uma tarefa. Por exemplo, uma receita de bolo é um algoritmo, pois explica como fazer o bolo passo-a-passo.

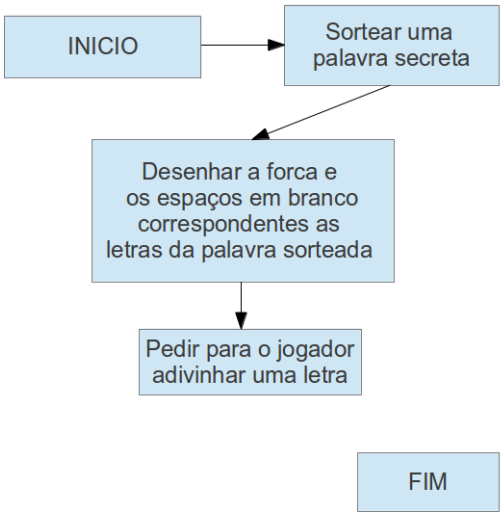

Figura 24: Desenhando os três primeiros passos do Fluxograma.

3. Mas o jogo não acaba após o jogador tentar adivinhar a letra. É preciso verificar se a letra está na palavra secreta ou não. Assim, precisamos adicionar duas novas caixas para o nosso fluxograma. Isto vai criar uma ramificação, como é mostrado na figura 25.

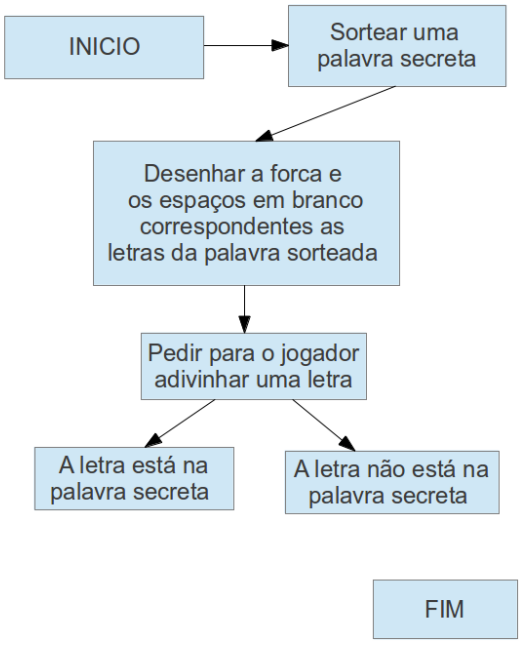

Figura 25: Ramificando o fluxograma.

4. Os processos de desenhar a forca e os espaços correspondentes a cada letra, pedir para que o jogador adivinhe a próxima letra e checar se a letra está ou não na palavra secreta devem ser repetidos várias vezes durante o jogo. Para isso precisamos criar um laço de repetição, como podemos ver na figura 26.

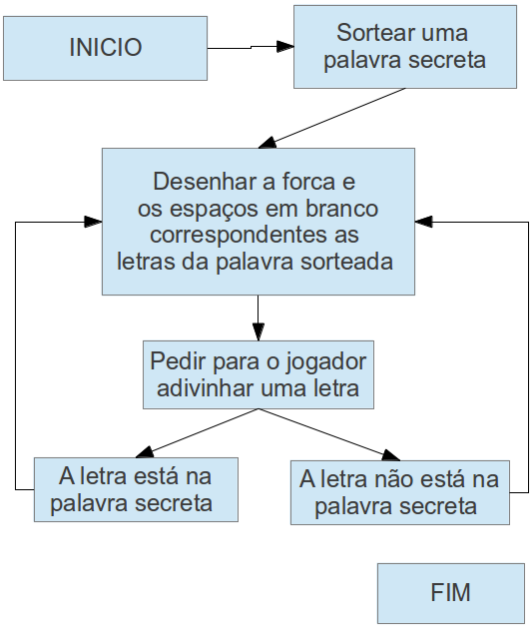

Figura 26: Criando os laços no fluxograma.

5. O jogador precisa ter o *feedback* do que está acontecendo no jogo, para isso será preciso imprimir as letras acertadas e a situação da forca após cada tentativa. Também é preciso verificar se o palpite do jogador já foi dado anteriormente no jogo, para isso fazemos algumas alterações na segunda intrução e incluiremos uma nova caixa no fluxograma para representar essa nova tarefa. Veja isso na figura 27.

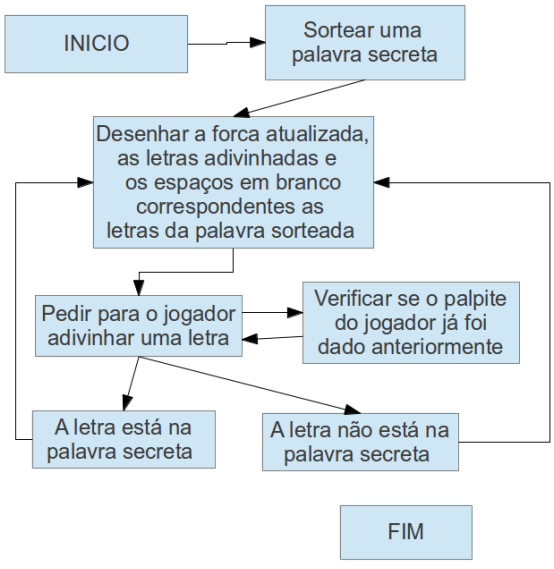

Figura 27: Alterando o segundo comando e adicionando a nova caixa.

6. Agora precisamos definir as codições de parada do jogo, caso contrário ele permanererá em um laço infinito. Após cada palpite e resposta, antes de finalizar o jogo, precisamos verificar se todas as letras foram adivinhadas, ou se todas as partes do boneco estão na forca, caso alguma dessas condições sejam satisfeitas o jogo deverá acabar com vitória ou derrota respectivamente. Ver figura 28

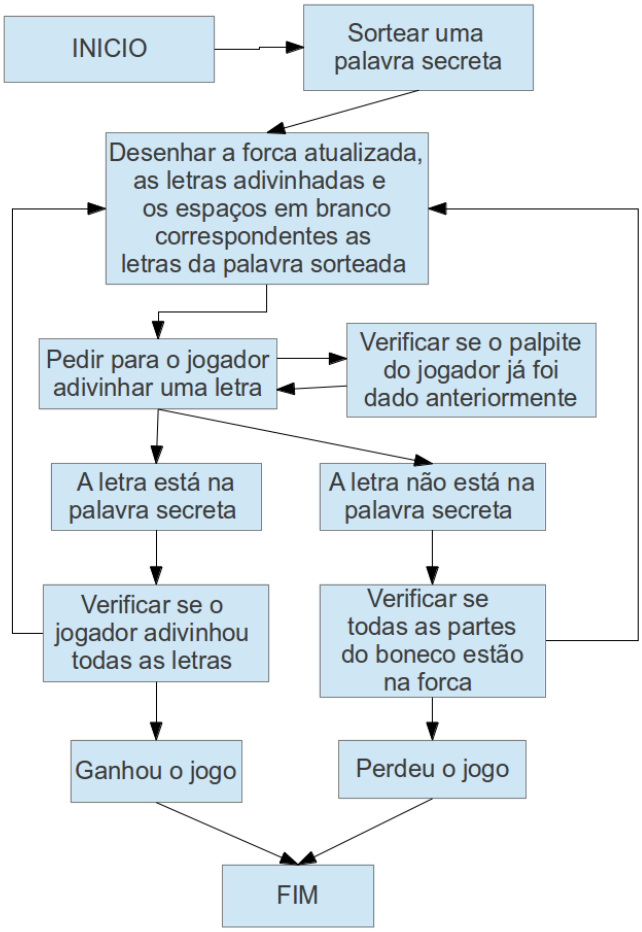

Figura 28: Incluindo as novas caixas.

7. Antes de finalizar, podemos também incluir a opção de iniciar um novo jogo. Para isso devemos incluir um novo laço iniciando na primeira instrução onde é sorteada a palavra secreta e terminando antes do fim do programa, concluindo assim o nosso fluxograma mostrado na figura 29.

Pode parecer muito trabalho elaborar um fluxograma antes de escrever o código. Mas é muito mais fácil fazer mudanças e encontrar problemas pensando em como vai funcionar o programa antes de escrever o código. Se pular esta parte e ir escrever o código, você pode descobrir problemas que exige que você altere o código já escrito. Toda vez que você altera o seu código, aumenta a chance de criar *bugs*. É melhor saber o que você quer construir antes de construir.

Quanto mais complexo for o programa, maior e mais carregado fica o fluxograma. Isso faz com que programadores mais experientes usem outras ferramentas para planejar seus programas com um maior nível de abstração[25].

[25] Habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes.

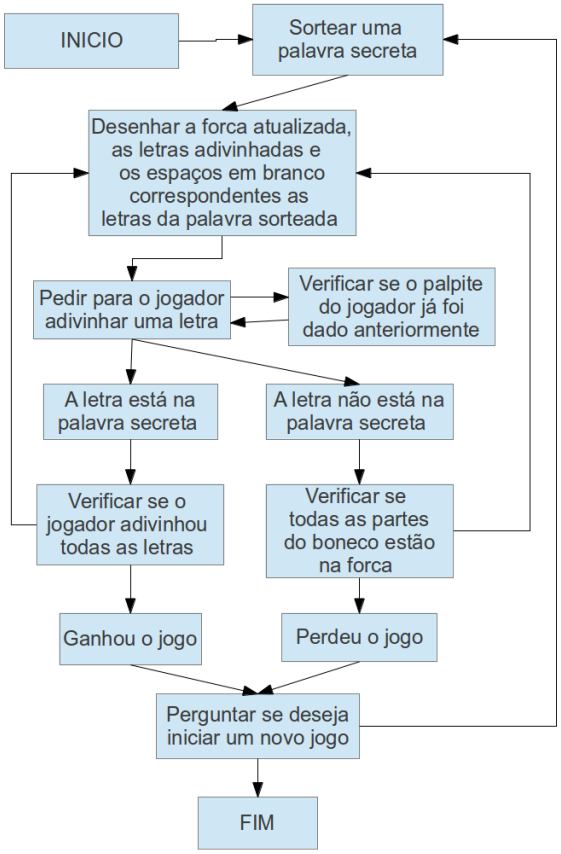

Figura 29: Fluxograma completo: Incluindo a nova caixa e o novo laço.

Referências Bibliográficas

1 MENDES, M. A. L. *Python Brasil - Lista de Exercícios*. 2012. Disponível em: <http://www-.python.org.br/wiki/ListaDeExerciciosl>. Acesso em: 13 ago. 2012.

2 SHAW, Z. A. *The Hard Way is Easier*. 2012. Disponível em: <http://learnpythonthehardway-.org/book/intro.html>. Acesso em: 15 jul. 2012.

3 SWEIGART, A. *Invent Your Own Games with Python*. 2010. Disponível em: <http:/-/inventwithpython.com/IYOCGwP_book1.pdf>. Acesso em: 20 jul. 2012.