Capítulo 2

O QUE É PYTHON

2.1 PYTHON

Python é uma linguagem de programação interpretada, orientada a objetos, de alto nível e com semântica dinâmica. A simplicidade do Python reduz a manutenção de um programa. Python suporta módulos e pacotes, que encoraja a programação modularizada e reuso de códigos.

É uma das linguagens que mais tem crescido devido sua compatibilidade (roda na maioria dos sistemas operacionais) e capacidade de auxiliar outras linguagens. Programas como *Dropbox*, *Reddit* e *Instagram* são escritos em Python. Python também é a linguagem mais popular para análise de dados e conquistou a comunidade científica.

Mas antes que você se pergunte o que cada uma dessas coisas realmente significa, vamos começar a desbravar o mundo Python e entender como funciona essa linguagem de programação que tem conquistado cada vez mais adeptos.

2.2 BREVE HISTÓRIA

Python foi criada em 1990 por Guido Van Rossum no Centro de Matemática Stichting (CWI, veja http://www.cwi.nl) na Holanda como uma sucessora da linguagem ABC. Guido é lembrado como o principal autor de Python, mas outros programadores ajudaram com muitas contribuições.

A linguagem ABC foi desenhada para uso de não programadores, mas logo de início mostrou certas

limitações e restrições. A maior reclamação dos primeiros alunos não programadores dessa linguagem era a presença de regras arbitrárias que as linguagens de programação haviam estabelecido tradicionalmente - muita coisa de baixo nível ainda era feita e não agradou o público.

Guido então se lançou na tarefa de criar uma linguagem de script simples que possuísse algumas das melhores propriedades da ABC. Listas Python, dicionários, declarações básicas e uso obrigatório de indentação - conceitos que aprenderemos neste curso - diferenciam Python da linguagem ABC. Guido pretendia que Python fosse uma segunda linguagem para programadores C ou C++ e não uma linguagem principal para programadores - o que mais tarde se tornou para os usuários de Python.

Em 1995, Guido continuou seu trabalho em Python na Corporation for National Research Initiatives (CNRI, veja http://www.cnri.reston.va.us/) in Reston, Virginia onde ele lançou outras versões da linguagem.

Em maio de 2000, Guido e o time principal de Python se mudaram para a BeOpen.com para formar o time BeOpen PythonLabs. Em outubro do mesmo ano, o time da PythonLabs se moveu para a Digital Creations (hoje, Zope Corporation, veja http://www.zope.org/). Em 2001, a Python Software Foundation (PSF, veja http://www.python.org/psf/), uma organização sem fins lucrativos, foi formada especialmente para manter a linguagem e hoje possui sua propriedade intelectual. A Zope Corporation é um membro patrocinador da PSF.

Todos os lançamentos de Python são de código aberto (veja http://www.opensource.org).

2.3 INTERPRETADOR

Você provavelmente já ouviu ou leu em algum lugar que Python é uma linguagem interpretada ou uma linguagem de script. Em certo sentido, também é verdade que Python é tanto uma linguagem interpretada quanto uma linguagem compilada. Um compilador traduz linguagem Python em linguagem de máquina - código Python é traduzido em um código intermediário que deve ser executado por uma máquina virtual conhecida como PVM (Python Virtual Machine). É muito similar ao Java - há ainda um jeito de traduzir programas Python em *bytecode* Java para JVM (Java Virtual Machine) usando a implementação Jython.

O interpretador faz esta 'tradução' em tempo real para código de máquina, ou seja, em tempo de execução. Já o compilador traduz o programa inteiro em código de máquina de uma só vez e então o executa, criando um arquivo que pode ser rodado (executável). O compilador gera um relatório de erros (casos eles existam) e o interpretador interrompe a tradução quando encontra um primeiro erro.

Em geral, o tempo de execução de um código compilado é menor que um interpretado já que o compilado é inteiramente traduzido antes de sua execução. Enquanto o interpretado é traduzido instrução por instrução. Python é uma linguagem interpretada mas, assim como Java, passa por um processo de compilação. Um código fonte Java é primeiramente compilado para um *bytecode* e depois

interpretado por uma máquina virtual.

Mas devemos compilar script Python? Como compilar? Normalmente, não precisamos fazer nada disso porque o Python está fazendo isso para nós, ou seja, ele faz este passo automaticamente. Na verdade, é o interpretador Pyhton, o CPython. A diferença é que em Java é mais clara essa separação, o programador compila e depois executa o código.

CPython é uma implementação da linguagem Python. Para facilitar o entendimento, imagine que é um pacote que vem com um compilador e um interpretador Python (no caso, uma Máquina Virtual Python) além de outras ferramentas para usar e manter o Python. CPython é a implementação de referência (a que você instala do site http://python.org).

2.4 QUAL VERSÃO UTILIZAR?

Para quem está começando, a primeira dúvida na hora da instalação é qual versão do Python devemos baixar. Aqui, depende do que se deseja fazer. O Python3 ainda possui algumas desvantagens em relação a versão 2, como o suporte de bibliotecas (que é mais reduzido) e pelo fato da maioria das distribuições Linux e o MacOS ainda utilizarem a versão 2 como padrão em seus sistemas. Porém, o Python3 é mais maduro e mais recomendável para o uso.

Existem casos que exigem o Python2 ao invés do Python3 como implementar algo em um ambiente que o programador não controla ou quando precisa utilizar algum pacote/módulo específico que não possui versão compatível com Python3. Vale ressaltar para quem deseja utilizar uma implementação alternativa do Python, como o IronPython ou Jython, que o suporte ao Python3 ainda é bastante limitado.

Atualmente existe a ferramenta ***2to3*** que permite que código Python3 seja gerado a partir de código Python2. Há também a ferramenta ***3to2***, que visa converter o código Python3 de volta ao código Python2. No entanto, é improvável que o código que faz uso intenso de recursos do Python3 seja convertido com sucesso.

JVM (Java Virtual Machine) e o *bytecode* do IronPython por uma Virtual Machine .NET.

Outra implementação que vem crescendo é o PyPy, uma implementação escrita em Python que possui uma Virtual Machine Python. É mais veloz do que o CPython e vem com a tecnologia JIT (Just In Time) que já "traduz" o código fonte em código de máquina.

O Compilador Python traduz um *programa.py* para *bytecode* - ele cria um arquivo correspondente chamado *programa.cpy*. Se quisermos ver o *bytecode* pelo terminal, basta usar o módulo *disassembler* (dis) que suporta análise do *bytecode* do CPython, desmontando-o. Você pode checar a documentação aqui: https://docs.python.org/3/library/dis.html.

2.7 PEP - O QUE SÃO E PRA QUE SERVEM

PEP, *Python Enhancement Proposals* ou Propostas para Melhoramento no Python, como o nome diz são propostas de aprimoramento ou de novas funcionalidades para a linguagem. Qualquer um pode escrever uma proposta e a comunidade Python testa, avalia e decide se deve ou não fazer parte da linguagem. Caso aprovado, o recurso é liberado para as próximas versões.

No site oficial do Python (https://www.python.org/) você pode checar todas as PEPs da linguagem. A PEP 0 é aquela que contém o índice de todas as propostas de aprimoramento do Python e pode ser acessada aqui: https://www.python.org/dev/peps.

Ao longo do curso, de acordo com a aprendizagem e uso de certas funcionalidades, citaremos algumas PEPs mais importantes.

2.8 ONDE USAR E OBJETIVOS

Python é uma linguagem de propósito geral. Muitas vezes precisamos lidar com tarefas laterais: buscar dados em um banco de dados, ler uma página na internet, exibir graficamente os resultados, criar

planilhas etc. E Python possui vários módulos prontos para realizar essas tarefas.

Por esse e outros motivos que Python ganhou grande popularidade na comunidade científica. Além disso, Python é extremamente legível e uma linguagem expressiva, ou seja, de fácil compreensão. As ciências, por outro lado, possuem raciocínio essencialmente complicado e seria um problema adicional para cientistas conhecerem, além de seu assunto de pesquisa, assuntos complexos de um programa de computador como alocação de memória, gerenciamento de recursos etc. O Python faz isso automaticamente de maneira eficiente, possibilitando o cientista se concentrar no problema estudado.

2.9 PRIMEIRO PROGRAMA

Vamos para nosso primeiro código! Um programa que imprime uma mensagem simples.

Para mostrar uma mensagem específica, fazemos:

```
print('Minha primeira aplicação Python!')
```

Certo, mas onde digitar esse comando? Como rodar uma instrução Python?

2.10 MODO INTERATIVO

Iremos, primeiro, aprender o **modo interativo** utilizando o terminal (Linux e MacOS) ou o prompt de comando (Windows) para rodar o programa acima. Abra o terminal e digite:

```
dev@caelum:~$ python3
```

Isso vai abrir o modo interativo do Python na versão 3.6 da linguagem, também chamado de console do Python. Após digitar este comando, as seguintes linhas irão aparecer no seu console:

```
Python 3.6.4 (default, Jan 28 2018, 00:00:00)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

A primeira linha indica que a versão utilizada do Python é a versão 3.6.4. A segunda indica o sistema operacional (no caso, o Linux). A terceira mostra algumas palavras chaves do interpretador para acessar algumas informações - digite alguma delas e aperte `ENTER` para testar.

O '>>>' indica que entramos no modo interativo do Python e basta começar a escrever os comandos. Vamos então escrever nosso primeiro programa Python:

```
>>> print('Minha primeira aplicação Python!')
```

Ao apertar `ENTER` , temos:

```
>>> print('Minha primeira aplicação Python!')
Minha primeira aplicação Python!
```

O **print()** é uma **função** do Python utilizada para imprimir alguma mensagem na tela. Mais detalhes

sobre funções são tratados em um capítulo específico desta apostila. Neste momento, entenda uma função como uma funcionalidade pronta que a linguagem fornece.

Uma mensagem deve estar delimitada entre aspas simples (') ou duplas ("), como feito no exemplo acima com a mensagem: 'Minha primeira aplicação Python!'. O interpretador, no modo interativo, já vai mostrar a saída deste comando no console, logo abaixo dele.

Mas e se um programa possuir 1.000 linhas de código? Teremos que digitar essas mil linhas todas as vezes para rodar o programa? Isso, obviamente, seria um problema. Existe outro modo de desenvolvimento no Python mais utilizado, que evita digitar um programa longo no console toda vez que precisar executá-lo.

2.11 MODO SCRIPT

O modo interativo é mais utilizado para testes, enquanto que o **modo script** é mais comumente utilizado na hora de desenvolver. No modo script, isolamos o código Python em um arquivo com extensão **.py**. Dessa maneira, o código é escrito uma única vez e executado pelo interpretador através do comando **python3** (ou o comando **python** se estiver utilizando o Python2).

Abra um editor de texto de sua preferência e escreva o programa anterior nele:

```
print('Minha primeira aplicação Python!')
```

Salve o arquivo como **programa.py**. Para executá-lo, abra o terminal, navegue até o diretório onde se encontra o arquivo **programa.py** e digite:

```
dev@caelum:~$ python3 programa.py
```

Ao apertar `ENTER`, vai aparecer no console:

```
dev@caelum:~$ python3 programa.py
```

Toda vez que um script Python é executado, um código *bytecode* é criado. Se um script Python é importado como um módulo, o *bytecode* vai armazenar seu arquivo **.pyc** correspondente.

Portanto, o passo seguinte não criará o arquivo *bytecode*, já que o interpretador vai verificar que não existe nenhuma alteração:

```
dev@caelum:~$ python programa.py
Minha primeira aplicação Python!
dev@caelum:~$
```

2.12 EXERCÍCIO: MODIFICANDO O PROGRAMA

1. Altere o programa para imprimir uma mensagem diferente.
2. Altere seu programa para imprimir duas linhas de código utilizando a função `print()`.
3. Sabendo que os caracteres `\n` representam uma quebra de linha, imprima duas linhas de texto usando uma única linha de código.

2.13 O QUE PODE DAR ERRADO?

Nem sempre as coisas acontecem como esperado. O Python tem uma sintaxe própria, um vocabulário próprio. Digitar algo que o interpretador não entende causará um erro no programa. Vejamos alguns exemplos:

Esquecer os parênteses

```
>>> print 'Minha primeira aplicação Python!'
Traceback (most recent call last):
File "<stdin>", line 1
    print 'Minha primeira aplicação Python!'
                      ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print('Minha primeira aplicação
Python!')?
```

Não se assuste com a mensagem. Vamos entender o que ela quer dizer. Na primeira linha aparece a palavra `Traceback` que significa algo como: "O que o programa estava fazendo quando parou porque algo de errado aconteceu?". É por este motivo que a mensagem `most recent call last` (chamada mais recente) é mostrada.

A `Traceback` faz referência a um arquivo - que é o nome do arquivo Python chamado acima pelo nome de `stdin` que possui métodos para leitura, onde o programa lê a entrada do teclado. O programa acusa que este erro está na primeira linha do programa: `File "<stdin>", line 1`.

Logo em seguida é mostrado exatamente a parte do código que gerou o erro: `print 'Minha primeira aplicação Python!'`. A próxima linha é a mensagem de erro: `SyntaxError`. Se você não

CAPÍTULO 3

VARIÁVEIS E TIPOS EMBUTIDOS

Neste capítulo vamos conhecer os tipos da biblioteca padrão do Python. Os principais tipos internos são números, sequências, mapas, classes, objetos e exceções, mas iremos focar primeiramente nos números e sequências de texto (*strings*). São objetos nativos da linguagem, recursos que já vêm prontos para uso e chamados de *built-ins*.

Neste início da aprendizagem, trabalharemos com o modo interativo, e ao final produziremos uma pequena aplicação em um script.

3.1 TIPOS EMBUTIDOS (BUILT-INS)

Um valor, como um número ou texto, é algo comum em um programa. Por exemplo, '*Hello, World!*', 1, 2, todos são valores. Estes valores são de diferentes tipos: 1 e 2 são números inteiros e '*Hello World!*' é um texto, também chamado de ***String***. Podemos identificar **strings** porque são delimitadas por aspas (simples ou duplas) - e é exatamente dessa maneira que o interpretador Python também identifica uma **string**.

A função `print()` utilizada no capítulo anterior também trabalha com inteiros:

```
>>> print(2)
2
```

Veja que aqui não é necessário utilizar aspas por se tratar de um **número**. Caso você não tenha certeza qual é o tipo de um valor, pode usar a função `type()` para checar:

```
>>> type('Hello World')
<class 'str'>

>>> type(2)
<class 'int'>
```

Strings são do tipo `str` (abreviação para *string*) e inteiros do tipo `int` (abreviação para *integer*). Ignore a palavra `class` por enquanto, teremos um capítulo especial para tratar dela. Veremos que funções como `type()` e `print()` também são tipos embutidos no Python.

Outro tipo que existe no Python são os números decimais que são do tipo `float` (ponto flutuante):

```
>>> type(3.2)
<class 'float'>
```

E qual será o tipo de valores como '2' e '3.2'? Eles se parecem com números mas são delimitados por aspas como *strings*. Utilize a função `type()` para fazer a verificação:

```
>>> type('2')
<class 'str'>

>>> type('3.2')
<class 'str'>
```

Como estão delimitados por aspas, o interpretador vai entender esses valores como *strings*, ou seja, como texto.

O Python também possui um tipo específico para números complexos. Números complexos são definidos por dois valores: a parte real e a parte imaginária. No Python é escrito na forma **real + imag j**. No caso, o número imaginário (definido pela raiz de -1 e chamado de 'i' na matemática) é designado pela letra `j` no Python. Por exemplo:

```
>>> 2 + 3j
>>> type(2 + 3j)
<class 'complex'>
```

2 é a parte real e 3 a parte imaginária do número complexo. Utilizando a função `type()`, podemos nos certificar que seu tipo é `complex`.

3.2 VARIÁVEIS

Podemos pedir para o Python lembrar de um valor que queiramos utilizar em outro momento do programa. O Python vai guardar este valor em uma **variável**. Variável é um nome que faz referência a um valor. É como uma etiqueta que colocamos naquele valor e quando precisarmos usar, chamamos pelo nome que foi dado na etiqueta.

Um comando de atribuição (o sinal de igualdade `=`) cria uma nova variável e atribui um valor a ela:

```
>>> mensagem = 'oi, python'
'oi, python'

>>> numero = 5
5

>>> pi = 3.14
3.14
```

Três atribuições foram feitas neste código. Atribuímos a variável `mensagem` uma *string*; a variável `numero` um inteiro e a variável `pi` um valor aproximado do número pi. No modo interativo, o interpretador mostra o resultado após cada atribuição.

Para recuperar esses valores, basta chamar pelos nomes das variáveis definidas anteriormente:

```
>>> mensagem
oi, python

>>> numero
5

>>> pi
3.14
```

Utilize a função `type()` para verificar seus tipos:

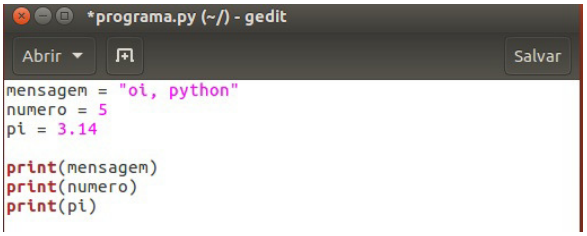
```
>>> type(mensagem)
<class 'str'>

>>> type(numero)
<class 'int'>

>>> type(pi)
<class 'float'>
```

3.3 PARA SABER MAIS: NOMES DE VARIÁVEIS

Programadores escolhem nomes para variáveis que sejam semânticos e que ao mesmo tempo documentem o código. Esses nomes podem ser bem longos, podem conter letras e números. É uma convenção entre os programadores Python começar a variável com letras minúsculas e utilizar o underscore (_) para separar palavras como: **meu_nome**, **numero_de_cadastro**, **telefone_residencial**. Esse padrão é chamado de ***snake case***. Variáveis também podem começar com underscore (_) mas deve ser evitado e utilizado em casos mais específicos.

Se nomearmos nossas variáveis com um nome ilegal, o interpretador vai acusar um erro de sintaxe:

```
>>> 1nome = 'python'
  File "<stdin>", line 1
  1nome = 'python'
    ^
SyntaxError: invalid syntax

>>> numero@ = 10
  File "<stdin>", line 1
```



Note que devemos utilizar a função `print()` para exibir os resultados na tela já que o modo script, diferente do modo interativo, não exibe os resultados após a declaração de variáveis.

Navegue até o diretório onde se encontra o arquivo **programa.py** e digite o comando no terminal:

```
dev@caelum:~$ python3 programa.py
```

Que vai gerar a saída:

```
dev@caelum:~$ python3 programa.py
oi, python
5
3.14
```


3.5 OPERADORES ARITMÉTICOS

Operadores são símbolos especiais que representam cálculos como adições e multiplicações. Para fazer cálculos com números utilizamos os operadores +, -, *, / e ** que representam, respectivamente, adição, subtração, multiplicação, divisão e potenciação.

Uma expressão é uma combinação de valores, variáveis e operadores, como `x + 17`, `1 + 1` etc. Quando digitamos uma expressão no modo interativo, o interpretador vai calcular e imprimir o resultado:

```
>>> 1 + 1
```

```
2

>>> 2 * 3
6
```

Também podemos usar variáveis:

```
>>> x = 1
>>> y = 3
>>> x + y
4

>>> x - y
-2

>>> x * y
3

>>> x / y
0.3333333333333333

>>> x ** y
1
```

Além dos operadores comentados, temos também o operador // que representa a divisão inteira:

```
>>> 7 // 2
3
```

E o operador módulo % que resulta no resto da divisão entre dois números inteiros:

```
>>> 7 % 3
1
```

7 divido por 3 é 2 e gera resto igual a 1. Esse operador é bem útil quando queremos checar se um número é divisível por outro.

Os principais operadores são:

Operação	Nome	Descrição
a + b	adição	Soma entre *a* e *b*
a - b	subtração	Diferença entre *a* e *b*
a * b	multiplicação	Produto entre *a* e *b*
a / b	divisão	Divisão entre *a* e *b*
a // b	divisão inteira	Divisão inteira entre *a* e *b*
a % b	módulo	Resto da divisão entre *a* e *b*
a ** b	exponenciação	*a* elevado a potência de *b*

3.6 STRINGS

O operador `+` também funciona com *strings* de uma maneira diferente dos números. Ele funciona

```
digite sua idade:
20
Seu nome é caelum e sua idade é 20
```

PARA SABER MAIS: A FUNÇÃO FORMAT()

A função `format()` faz parte de um conjunto de funções de formatação de *strings* chamada **Formatter**. Para mais detalhes, acesse a documentação: https://docs.python.org/3/library/string.html#string.Formatter.

Há outras funções de formatação e a `format()` é a principal delas e a mais utilizada. Com ela, podemos passar qualquer tipo de parâmetro, além de ser extremamente útil para formatar números passando seu *format code*. Por exemplo, podemos arredondar o número flutuante 245.2346 para duas casas decimais através do código de formatação `:.2f`:

```
>>> x = 245.2346
>>> print('{:.2f}'.format(x))
245.23
```

O `:.2f` diz que queremos apenas duas casas decimais para a variável `x`. Na documentação oficial do Python você acessa os códigos de formatação ou através da PEP 3101: https://www.python.org/dev/peps/pep-3101/.

3.8 CONSTANTES

O Python possui poucas constantes embutidas. As mais utilizadas são **True**, **False** e **None**. Essas também são palavras chaves do Python, portanto palavras reservadas que não podemos utilizar como nomes de variáveis.

O operador `==` checa se o conteúdo das variáveis são iguais e seu comportamento pode variar de interpretador para interpretador - o exemplo acima é o comportamento padrão do CPython. Já o operador `is` checa se `a` e `b` são o mesmo **objeto**. Falaremos de objetos em um outro capítulo, mas é importante ter em mente que tudo em Python é um objeto e cada objeto possui uma referência na memória. O operador `is` vai checar exatamente se `x` e `y` são o mesmo objeto, ou seja, se possuem a mesma referência.

3.9 COMANDO IF

E se quisermos apresentar uma mensagem diferente para o usuário dependendo do valor de entrada? Vamos atribuir um valor para uma variável `numero` e pedir para o usuário entrar com um valor. Devemos verificar se os valores são iguais como um jogo de adivinhação em que o usuário deve adivinhar o número definido.

```
numero = 42
chute = input('Digite um número: ')
```

Até aqui, nenhuma novidade. Agora, devemos mostrar a mensagem "Você acertou" caso o `numero` seja igual ao `chute`, e "Você errou" caso o `numero` seja diferente do `chute`. Em português, seria assim:

```
Se chute igual a número: "Você acertou"
Se chute diferente de número: "Você errou"
```

Ou melhor:

```
Se chute igual a número: "Você acertou"
Senão: "Você errou"
```

Podemos traduzir isso para código Python. O Python possui o operador condicional para representar a palavra **se** que é o **if** e a palavra **senão** que é o **else**. A sintaxe ficaria:

```
if chute == numero:
print('Você acertou')
else:
print('Você errou')
```

Este código ainda não funciona porque o Python entende as instruções `if` e `else` como blocos e os blocos devem seguir uma **indentação**. Como `print('Você acertou')` é a instrução que deve ser executada caso a verificação do `if` seja verdadeira, devemos ter um recuo para a direita em quatros espaços:

```
if chute == numero:
   print('Você acertou')
else:
   print('Você errou')
```

Caso contrário, o interpretador vai acusar erro de sintaxe. Dessa maneira, o código fica mais legível

(número).

Para funcionar, precisamos **converter** a *string* "42" para um número inteiro. O **int** também funciona como uma função (mais para frente entenderemos que não é realmente uma função) que pode receber uma *string* e retornar o inteiro correspondente:

```
>>> numero_em_texto = '12'
'12'
>>> type(numero_em_texto)
<class 'str'>
>>> numero = int(numero_em_texto)
12
>>> type(numero)
<class 'int'>
```

Mas devemos tomar cuidado, nem toda *string* pode ser convertida para um número inteiro:

```
>>> texto = 'caelum'
>>> numero = int(texto)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'caelum'
```

O interpretador acusa um **ValueError** dizendo que o valor passado para `int()` é inválido, ou seja, é um texto que não representa um número inteiro.

3.11 O COMANDO ELIF

Podemos melhorar ainda mais o jogo: caso o chute não seja igual ao número secreto, podemos dar uma pista para o usuário se ele foi maior ou menor do que o chute inicial. Ou seja, devemos acrescentar esse tratamento caso o usuário erre o chute:

```
Se chute = número:
    "Você acertou!"
Senão:
    Se chute maior do que número_secreto:
        "Você errou! O seu chute foi maior que o número secreto"
```

9. Vamos apresentar uma pista para o usuário e imprimir uma mensagem dizendo se o chute foi maior ou menor do que o número secreto. Para isso usaremos o `elif`:

```
if (numero_secreto == chute):
    print('Você acertou!')
elif (chute > numero_secreto):
    print('Você errou! O seu chute foi maior que o número secreto')
elif (chute < numero_secreto):
    print('Você errou! O seu chute foi menor que o número secreto')
```

10. Agora vamos melhorar a legibilidade do código extraindo as condições para variáveis:

```
acertou = chute == numero_secreto
maior = chute > numero_secreto
menor = chute < numero_secreto

if(acertou):
    print('Você acertou!')
elif(maior):
    print('Você errou! O seu chute foi maior que o número secreto')
elif(menor):
    print('Você errou! O seu chute foi menor que o número secreto')
```

11. Rode o programa e teste com todas as situações possíveis.

3.13 COMANDO WHILE

Queremos dar mais de uma oportunidade para o usuário tentar acertar o número secreto, já que é um jogo de adivinhação. A primeira ideia é repetir o código, desde a função `input()` até o bloco do `elif`. Ou seja, para cada nova tentativa que quisermos dar ao usuário, copiaríamos esse código novamente.

Só que copiar código sempre é uma má prática, queremos escrever o código apenas uma vez. Se queremos repetir o código, fazemos um laço, ou um loop, que deve repetir a instrução dentro de bloco **enquanto** ela for verdadeira. O laço que devemos fazer é:

```
Enquanto ainda há tentativas, faça:
    chute_str = input('Digite o seu número: ')
    print('Você digitou: ', chute_str)
    chute = int(chute_str)

    acertou = numero_secreto == chute
    maior = chute > numero_secreto
    menor = chute < numero_secreto

    if (acertou):
        print('Você acertou!')
    elif (maior):
        print('Você errou! O seu chute foi maior que o número secreto')
    elif (menor):
        print('Você errou! O seu chute foi menor que o número secreto')

    print('Fim do Jogo!')
```

```
        # restante do código
```

8. Teste o programa e veja se tudo está funcionando como o esperado.

3.15 COMANDO FOR

Ainda no código do jogo da adivinhação, implementamos o *loop* `while`, no qual temos uma variável `rodada` que começa com o valor 1, e é incrementada dentro do *loop*, que por sua vez tem uma condição de entrada, que é a `rodada` ser menor ou igual ao total de tentativas, que é 3.

Ou seja, a `rodada` tem um valor inicial, que é 1, e vai até 3. Fazemos um laço começando com um valor inicial, até um valor final, sempre incrementando esse valor a cada iteração. Mas se esquecermos de incrementar a rodada, entramos em um *loop* infinito.

Em casos como esse, existe um outro *loop* que simplifica essa ideia de começar com um valor e incrementá-lo até chegar em um valor final: o *loop* **for**.

Para entender o *loop*, ou laço **for**, podemos ir até o console do Python para ver o seu funcionamento. A ideia é definirmos o valor inicial e o valor final, que o *loop* o incrementa automaticamente. Para isto utilizamos a função embutida `range()`, passando-os por parâmetro, definindo assim a série de valores. A sintaxe é a seguinte:

```
Para variável em uma série de valores:
    Faça algo
```

Isso, em Python, pode ficar assim:

```
for rodada in range(1, 10):
```

O `range(1, 10)` vai gerar o intervalo de números inteiros de 1 a 9. Na primeira iteração, o valor da variável `rodada` será 1, depois 2 e até chegar ao valor final da função `range()` menos 1, isto é, o segundo parâmetro da função não é inclusivo. No exemplo acima, a série de valores é de 1 a 9. Podemos confirmar isso imprimindo o valor da variável `rodada` no console do Python:

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

Com a função `range()`, podemos definir um *step* (um passo), que é o intervalo entre os elementos. Por padrão, o *step* tem valor igual a 1, mas podemos alterar este valor passando um terceiro parâmetro

Capítulo 5

ESTRUTURA DE DADOS

No capítulo passado criamos o jogo da adivinhação, agora vamos criar o jogo da Forca. Vamos começar com os conhecimentos que temos até aqui para estruturarmos o nosso jogo. Vamos criar um arquivo chamado *forca.py* na pasta *jogos*:

```
|_ home
|_ jogos
    |_ adivinhacao.py
    |_ forca.py
```

Como no jogo da adivinhação, devemos adivinhar uma palavra secreta, então nada mais justo que defini-la em uma variável. Por enquanto, a palavra será fixa, com o valor **banana**:

```
print('**********************************')
print('***Bem vindo ao jogo da Forca!***')
print('**********************************')

palavra_secreta = 'banana'

print('Fim do jogo')
```

Mais à frente deixaremos essa palavra secreta mais dinâmica.

Como estamos tratando de um jogo da forca, o usuário deve acertar uma palavra e chutar letras. Além disso, precisa saber se o usuário acertou ou errou e saber se foi enforcado ou não. Então precisaremos de 2 variáveis booleanas `enforcou` e `acertou` para guardar esta informação e o jogo continuará até uma delas for `True` ; para tal acrescentamos um laço `while` :

```
acertou = False
enforcou= False

while(not acertou and not enforcou):
    print('Jogando...')
```

O usuário do jogo também vai chutar letras. Além disso, se o chute for igual a uma letra contida na `palavra_secreta` , quer dizer que o usuário encontrou uma letra. Podemos utilizar um laço `for` para tratar isso, assim como fizemos no jogo da adivinhação para apresentar as tentativas e rodadas:

```
while(not acertou and not errou):
    chute = input('Qual letra?')

    posicao = 0
    for letra in palavra_secreta:
        if (chute == letra):
            print('Encontrei a letra {} na posição {}'.format(letra, posicao))
```

```
if(acertou):
    print('Você ganhou!!')
else:
    print('Você perdeu!!')
print('Fim do jogo')
```


5.2 SEQUÊNCIAS

Desenvolvemos um novo jogo e conhecemos uma nova estrutura, as listas. Uma lista é denominada por uma sequência de valores, e o Python possui outros tipos de dados que também são sequências. Neste momento, conheceremos um pouco de cada uma delas.

Sequências são *containers*, um tipo de dado que contém outros dados. Existem três tipos básicos de sequência: `list` (lista) , `tuple` (tupla) e `range` (objeto de intervalo). Outro tipo de sequência famoso que já vimos são as *strings* que são sequências de texto.

Sequências podem ser mutáveis ou imutáveis. Sequências imutáveis não podem ter seus valores modificados. Tuplas, *strings* e *ranges* são sequências imutáveis, enquanto listas são sequências mutáveis.

As operações na tabela a seguir são suportadas pela maioria dos tipos de sequência, mutáveis e imutáveis. Na tabela abaixo, s e t são sequências do mesmo tipo, n , i , j e k são inteiros e x é um objeto arbitrário que atende a qualquer tipo e restrições de valor impostas por s .

Operação	Resultado
x in s	True se um item de s é igual a *x*
x not in s	False se um item de s é igual a *x*
s + t	Concatenação de s e s
s *n ou n* s	Equivalente a adicionar s a si mesmo n vezes
s[i]	Elemento na posição i de s

s[i:j]	Fatia s de i para j
s[i:j:k]	Fatia s de i para j com o passo k
len(s)	Comprimento de s
min(s)	Menor item de s
max(s)	Maior item de s
s.count(x)	Número total de ocorrências de x em s

- **Listas**

Uma lista é uma sequência de valores onde cada valor é identificado por um índice iniciado por 0. São similares a *strings* (coleção de caracteres) exceto pelo fato de que os elementos de uma lista podem ser de qualquer tipo. A sintaxe é simples, listas são delimitadas por colchetes e seus elementos separados por vírgula:

```
>>> lista1 = [1, 2, 3, 4]
>>> lista1
[1, 2, 3, 4]
>>>
>>>lista2 = ['python', 'java', 'c#']
>>> lista2
['python', 'java', 'c#']
```

O primeiro exemplo é uma lista com 4 inteiros, o segundo é uma lista contendo três *strings*. Contudo, listas não precisam conter elementos de mesmo tipo. Podemos ter listas heterogêneas:

```
>>> lista = [1, 2, 'python', 3.5, 'java']
>>> lista
[1, 2, 'python', 3.5, 'java']
```

Nossa *lista* possui elementos do tipo `int`, `float` e `str`. Se queremos selecionar um elemento específico, utilizamos o operador `[ ]` passando a posição:

```
>>> lista = [1, 2, 3, 4]
>>> lista[0]
1
>>> lista[1]
2
>>> lista[2]
3
>>> lista[3]
4
>>> lista[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Quando tentamos acessar a posição 4 fazendo `lista[4]`, aparece o erro `IndexError`, dizendo que a lista excedeu seu limite já que não há um quinto elemento (índice 4).

O Python permite passar valores negativos como índice que vai devolver o valor naquela posição de

Quando é necessário armazenar uma coleção de dados que não possa ser alterada, prefira usar tuplas a listas. Outra vantagem é que tuplas podem ser usadas como chaves de dicionários, que discutiremos nas seções adiante.

Tuplas são frequentemente usadas em programas Python. Um uso bastante comum são em funções que recebem múltiplos valores. As tuplas implementam todas as operações de sequência comuns.

- **Range**

O range é um tipo de sequência imutável de números, sendo comumente usado para *looping* de um número específico de vezes em um comando `for` já que representam um intervalo. O comando **range** gera um valor contendo números inteiros sequenciais, obedecendo a sintaxe:

```
range(inicio, fim)
```

O número finalizador, o *fim*, não é incluído na sequência. Vejamos um exemplo:

```
>>> sequencia = range(1, 3)
>>> print(sequencia)
range(1, 3)
```

O range não imprime os elementos da sequência, ele apenas armazena seu início e seu final. Para imprimir seus elementos precisamos de um laço `for` :

```
>>> for valor in range(1, 3):
...     print(valor)
...
1
2
```

Observe que ele não inclui o segundo parâmetro da função range na sequência. Outra característica deste comando é a de poder controlar o passo da sequência adicionando um terceiro parâmetro, isto é, a variação entre um número e o seu sucessor:

```
>>> for valor in range(1, 10, 2):
...     print(valor)
...
1
3
5
7
9
```

Os intervalos implementam todas as operações de seqüência comuns, exceto concatenação e repetição (devido ao fato de que objetos de intervalo só podem representar sequências que seguem um padrão estrito e a repetição e a concatenação geralmente violam esse padrão).

5.3 CONJUNTOS

O Python também inclui um tipo de dados para conjuntos. Um conjunto, diferente de uma sequência,

é uma coleção **não ordenada** e que **não admite elementos duplicados**.

Chaves ou a função `set()` podem ser usados para criar conjuntos.

```
>>> frutas = {'laranja', 'banana', 'uva', 'pera', 'laranja', 'uva', 'abacate'}
>>> frutas
>>> {'uva', 'abacate', 'pera', 'banana', 'laranja'}
>>> type(frutas)
<class 'set'>
```

Usos básicos incluem testes de associação e eliminação de entradas duplicadas. Os objetos de conjunto também suportam operações matemáticas como união, interseção, diferença e diferença simétrica. Podemos transformar um texto em um conjunto com a frunção `set()` e testar os operações:

```
>>> a =  set('abacate')
>>> b = set('abacaxi')
>>> a
{'a', 'e', 'c', 't', 'b'}
>>> b
{'a', 'x', 'i', 'c', 'b'}
>>> a - b                              # diferença
{'e', 't'}
>>> a | b                              # união
{'c', 'b', 'i', 't', 'x', 'e', 'a'}
>>> a & b                              # interseção
{'a', 'c', 'b'}
>>> a ^ b                              # diferença simétrica
{'i', 't', 'x', 'e'}
```

Note que para criar um conjunto vazio você tem que usar `set()`, não `{}`; o segundo cria um dicionário vazio, uma estrutura de dados que discutiremos na próxima seção.

```
>>> a = set()
>>> a
set()
>>> b = {}
>>> b
{}
>>> type(a)
<class 'set'>
>>> type(b)
<class 'dict'>
```

5.4 DICIONÁRIOS

Vimos que `list`, `tuple`, `range` e `str` são sequências ordenadas de objetos, e *sets* são coleções de elementos não ordenados. Dicionário é outra estrutura de dados em Python e seus elementos, sendo estruturadas de forma não ordenada assim como os conjuntos. Ainda assim, essa não é a principal diferença com as listas. Os dicionários são estruturas poderosas e muito utilizadas, já que podemos acessar seus elementos através de chaves e não por sua posição. Em outras linguagens, este tipo é conhecido como "matrizes associativas".

Qualquer chave de um dicionário é associada (ou mapeada) a um valor. Os valores podem ser

CAPÍTULO 6

FUNÇÕES

Objetivos:

- entender o conceito de função
- saber e usar algumas funções embutidas da linguagem
- criar uma função

6.1 O QUE É UMA FUNÇÃO?

O conceito de função é um dos mais importantes na matemática. Em computação, uma função é uma sequência de instruções que computa um ou mais resultados que chamamos de parâmetros. No capítulo anterior, utilizamos algumas funções já prontas do Python, como o **print()**, **input()**, **format()** e **type()**.

Também podemos criar nossas próprias funções. Por exemplo, quando queremos calcular a razão do espaço pelo tempo, podemos definir uma função recebendo estes parâmetros:

```
f(espaco, tempo) = espaco/tempo
```

Essa razão do espaço pelo tempo é o que chamamos de velocidade média na física. Podemos então dar este nome a nossa função:

```
velocidade(espaco, tempo) = espaco/tempo
```

Se um carro percorreu uma distância de 100 metros em 20 segundos, podemos calcular sua velocidade média:

```
velocidade(100, 20) = 100/20 =     5 m/s
```

O Python permite definirmos funções como essa da velocidade média. A sintaxe é muito parecida com a da matemática. Para definirmos uma função no Python, utilizamos o comando **def**:

```
def velocidade(espaco, tempo):
    pass
```

Logo após o `def` vem o nome da função, e entre parênteses vêm os seus parâmetros. Uma função também tem um escopo e um bloco de instruções onde colocamos os cálculos, onde estes devem seguir a identação padrão do Python (4 espaços a direita).

Como nossa função ainda não faz nada, utilizamos a palavra chave **pass** para dizer ao interpretador que definiremos os cálculos depois. A palavra `pass` não é usada apenas em funções, podemos usar em

qualquer bloco de comandos como nas instruções `if` , `while` e `for` , por exemplo.

Vamos substituir a palavra `pass` pelos cálculos que nossa função deve executar:

```
def velocidade(espaco, tempo):
    v = espaco/tempo
    print('velocidade: {} m/s'.format(v))
```

Nossa função faz o cálculo da velocidade média e utiliza a função `print()` do Python para imprimi-lo na tela. Vamos testar nossa função:

```
velocidade(100, 20)

#velocidade: 5 m/s
```

De maneira geral, uma função é um estrutura para agrupar um conjunto de instruções que podem ser reutilizadas. Agora qualquer parte do nosso programa pode chamar a função velocidade quando precisar calcular a velocidade média de um veículo, por exemplo. E podemos chamá-la mais de uma vez, o que significa que não precisamos escrever o mesmo código novamente.

Funções são conhecidas por diversos nomes em linguagens de programação como subrotinas, rotinas, procedimentos, métodos e subprogramas.

Podemos ter funções sem parâmetros. Por exemplo, podemos ter uma função que diz 'oi' na tela:

```
def diz_oi():
   print("oi")
diz_oi()

#oi
```


6.2 PARÂMETROS DE FUNÇÃO

Um conjunto de parâmetros consiste em uma lista com nenhum ou mais elementos que podem ser obrigatórios ou opcionais. Para um parâmetro ser opcional, o mesmo é atribuído a um valor padrão

(default) - o mais comum é utilizar **None**. Por exemplo:

```
def dados(nome, idade=None):
    print('nome: {}'.format(nome))
    if(idade is not None):
        print('idade: {}'.format(idade))
    else:
        print('idade: não informada')
```

O código da função acima recebe uma idade como parâmetro e faz uma verificação com uma instrução **if**: se a idade for diferente de *None* ela vai imprimir a idade, caso contrário vai imprimir *idade não informada*. Vamos testar passando os dois parâmetros e depois apenas o nome:

```
dados('joão', 20)

#nome: joão
#idade: 20
```

Agora passando apenas o nome:

```
dados('joão')

#nome: joão
#idade: não informada
```

E o que acontece se passarmos apenas a idade?

```
dados(20)

#nome: 20
#idade: não informada
```

Veja que o Python obedece a ordem dos parâmetros. Nossa intenção era passar o número 20 como `idade`, mas o interpretador vai entender que estamos passando o `nome` porque não avisamos isso à ele. Caso queiramos passar apenas a `idade`, devemos especificar o parâmetro:

```
dados(idade=20)
 File "<stdin>", line 1, in <module>
TypeError: dados() missing 1 required positional argument: 'nome'
```

O interpretador irá acusar um erro, já que não passamos o atributo obrigatório `nome`.

6.3 FUNÇÃO COM RETORNO

E se ao invés de apenas mostrar o resultado, quisermos utilizar a velocidade média para fazer outro cálculo, como calcular a aceleração? Da maneira como está, nossa função `velocidade()` não consegue utilizar seu resultado final para cálculos.

```
Exemplo:
aceleracao = velocidade(parametros) / tempo
```

Para conseguirmos este comportamento, precisamos que nossa função **retorne** o valor calculado por ela. Em Python e em outras linguagens de programação, isto é alcançado através do comando **return**:

```
def velocidade(espaco, tempo):
    v = espaco/tempo
    return v
```

Testando:

```
print(velocidade(100, 20))
#5.0
```

Ou ainda, podemos atribuí-la a uma variável:

```
resultado = velocidade(100, 20)
print(resultado)
#5.0
```

E conseguimos utilizá-la no cálculo da aceleração:

```
aceleracao = velocidade(100, 20)/20
print(aceleracao)
#0.25
```

Uma função pode conter mais de um comando **return**. Por exemplo, nossa função `dados()` que imprime o nome e a idade, pode agora retornar uma *string*. Repare que, neste caso, temos duas situações possíveis: a que a idade é passada por parâmetro e a que ela não é passada. Aqui, teremos dois comandos **return**:

```
def dados(nome, idade=None):
    if(idade is not None):
        return ('nome: {} \nidade: {}'.format(nome, idade))
    else:
        return ('nome: {} \nidade: não informada'.format(nome))
```

Apesar da função possuir dois comandos **return**, ela tem apenas um retorno -- vai retornar um ou o outro. Quando a função encontra um comando **return**, ela não executa mais nada que vier depois dele dentro de seu escopo.

6.4 RETORNANDO MÚLTIPLOS VALORES

Apesar de uma função executar apenas um retorno, em Python podemos retornar mais de um valor. Vamos fazer uma função calculadora que vai retornar os resultados de operações básicas entre dois números: adição(+) e subtração(-), nesta ordem.

Para retornar múltiplos valores, retornamos os resultados separados por vírgula:

```
def calculadora(x, y):
    return x+y, x-y

print(calculadora(1, 2))

#(3, -1)
```

Qual será o tipo de retorno desta função? Vamos perguntar ao interpretador através da função type:

```
print(type(calculadora(1,2)))
```

```
if escolha == 1:
    # Jogar adivinhação
elif escolha == 2:
    # Jogar forca
```

Chame a função `jogar()` do módulo do jogo escolhido:

```
# Importa módulos e recebe a escolha do usuário (código omitido)

if escolha == 1:
    adivinhacao.jogar()
elif escolha == 2:
    forca.jogar()
```

Agora toda vez que o usuário quiser jogar um de nossos jogos, ele poderá escolher por meio do menu criado.

6.10 MÓDULOS E O COMANDO IMPORT

Ao importar o arquivo forca.py, estamos importando um **módulo** de nosso programa, que nada mais é do que um arquivo. Vamos verificar o tipo de `forca`:

```
print(type(forca))
<class 'module'>
```

Veja que o tipo é um **módulo**. Antes de continuarmos com nosso jogo, vamos aprender um pouco mais sobre arquivos e módulos. Vamos melhorar ainda mais nosso jogo da Forca e utilizar o que aprendemos de funções para organizar nosso código.

Capítulo 8

ORIENTAÇÃO A OBJETOS

Considere um programa para um banco financeiro. É fácil perceber que uma entidade importante para o nosso sistema será uma conta. Primeiramente, suponha que você tem uma conta nesse banco com as seguintes características: titular, número, saldo e limite. Vamos começar inicializando essas características:

```
>>> numero = '123-4'
>>> titular = "João"
>>> saldo = 120.0
>>> limite = 1000.0
```

E se a necessidade de representar mais de uma conta surgir? Vamos criar mais uma:

```
>>> numero1 = '123-4'
>>> titular1 = "João"
>>> saldo1 = 120.0
>>> limite1 = 1000.0
>>>
>>> numero2 = '123-5'
>>> titular2 = "José"
>>> saldo2 = 200.0
>>> limite2 = 1000.0
```

Nosso banco pode vir a crescer e ter milhares de contas e, da maneira que está o programa, seria muito trabalhoso dar manutenção.

E como utilizar os dados de uma determinada conta em outro arquivo? Podemos utilizar a estrutura do dicionário que aprendemos anteriormente e agrupar essas características. Isso vai ajudar a acessar os dados de uma conta específica:

```
conta = {"numero": '123-4', "titular": "João", "saldo": 120.0, "limite": 1000.0}
```

Agora é possível acessar os dados de uma conta pelo nome da chave:

```
>>> conta['numero']
'123-4'
>>> conta['titular']
'João'
```

Para criar uma segunda conta, crie outro dicionário:

```
conta2 = {"numero": '123-5', "titular": "José", "saldo": 200.0, "limite": 1000.0}
```

Avançamos em agrupar os dados de uma conta, mas ainda precisamos repetir seguidamente essa linha de código a cada conta criada. Podemos isolar esse código em uma função responsável por criar

```
def deposita(conta, valor):
    conta['saldo'] += valor
```

5. Crie outra função chamada `saca()` que recebe como argumento uma `conta` e um `valor`. Dentro da função, subtraia o `valor` do `saldo` da conta:

```
def saca(conta, valor):
    conta['saldo'] -= valor
```

6. E por fim, crie uma função chamada `extrato()`, que recebe como argumento uma `conta` e imprime o `numero` e o `saldo`:

```
def extrato(conta):
    print("numero: {} \nsaldo: {}".format(conta['numero'], conta['saldo']))
```

1. Navegue até a pasta `oo`, digite os comandos no arquivo *teste_conta.py* e teste as funcionalidades:

```
conta = cria_conta('123-7', 'João', 500.0, 1000.0)
deposita(conta, 50.0)
extrato(conta)

#numero: '123-7'
#saldo: 550.0

saca(conta, 20.0)
extrato(conta)

#numero: '123-7'
#saldo 530.0
```

2. (Opcional) Acrescente uma documentação para o seu módulo `teste_conta.py` e utilize a função `help()` para testá-la.

Neste exercício criamos uma conta e juntamos suas características (número, titular, limite, saldo) e funcionalidades (sacar, depositar, tirar extrato) num mesmo arquivo. Mas o que fizemos até agora foi baseado no conhecimento procedural que tínhamos do Python3.

Por mais que tenhamos agrupado os dados de uma conta, essa ligação é frágil no mundo procedural e se mostra limitada. Precisamos pensar sobre o que escrevemos para não errar. O paradigma orientado a objetos vem para sanar essa e outras fragilidades do paradigma procedural que veremos a seguir.

8.3 CLASSES E OBJETOS

Ninguém deveria ter acesso ao saldo diretamente. Além disso, nada nos obriga a validar esse valor e podemos esquecer disso cada vez que utilizá-lo. Nosso programa deveria obrigar o uso das funções `saca()` e `deposita()` para alterar o saldo e não permitir alterar o valor diretamente:

```
conta3['saldo'] = 100000000.0
```

ou então:

```
conta3['saldo'] = -3000.0
```

Devemos manipular os dados através das funcionalidades `saca()` e `deposita()` e proteger os dados da conta. Pensando no mundo real, ninguém pode modificar o saldo de sua conta quando quiser, a não ser quando vamos fazer um saque ou um depósito. A mesma coisa deve acontecer aqui.

Para isso, vamos primeiro entender o que é `classe` e `objeto`, conceitos importantes do paradigma **orientado a objetos** e depois veremos como isso funciona na prática.

Quando preparamos um bolo, geralmente seguimos uma receita que define os ingredientes e o modo de preparação. A nossa conta é um objeto concreto assim como o bolo que também precisa de uma receita pré-definida. E a "receita" no mundo OO recebe o nome de **classe**. Ou seja, antes de criarmos um objeto, definiremos uma classe.

Outra analogia que podemos fazer é entre o projeto de uma casa (a planta da casa) e a casa em si. O projeto é a **classe** e a casa, construída a partir desta planta, é o **objeto**. O projeto da conta, isto é, a definição da conta, é a classe. O que podemos construir (instanciar) a partir dessa classe, as contas de verdade, damos o nome de objetos.

Pode parecer óbvio, mas a dificuldade inicial do paradigma da orientação a objetos é justamente saber distinguir o que é classe e o que é objeto. É comum o iniciante utilizar, obviamente de forma errada, essas duas palavras como sinônimos.

O próximo passo será criar nossa classe `Conta` dentro de um novo arquivo Python, que receberá o nome de *conta.py*. Criar uma classe em Python é extremamente simples em termos de sintaxe. Vamos começar criando uma classe vazia. Depois criaremos uma instância, um objeto dessa classe, e utilizaremos a função `type()` para analisar o resultado:

```
class Conta:
    pass

    #Em outro arquivo:
    from conta import Conta

    conta = Conta()
    print(type(conta))
    #<class '__main__.Conta'>
```

Vemos pois o módulo onde se encontra a classe `Conta` é no módulo principal, ou seja, na nossa `__main__`. Agora, temos uma classe `Conta`.

Como Python é uma linguagem dinâmica, podemos modificar esse objeto `conta` em tempo de execução. Por exemplo, podemos acrescentar **atributos** a ele:

```
    conta.titular = "João"
    print(conta.titular)
    #'João'

    conta.saldo = 120.0
```

```
print(conta.saldo)
#120.0
```

Mas o problema do código é que ainda não garantimos que toda instância de `Conta` tenha um atributo `titular` ou `saldo`. Portanto, queremos uma forma padronizada da conta de maneira que possamos criar objetos com determinadas configurações iniciais.

Em linguagens orientadas a objetos, existe uma maneira padronizada de criar atributos em um objeto. Geralmente fazemos isso através de uma função construtora - algo parecido com nossa função `cria_conta()` do exercício anterior.

8.4 CONSTRUTOR

Em Python, alguns nomes de métodos estão reservados para o uso da própria linguagem. Um desses métodos é o `__init__()` que vai inicializar o objeto. Seu primeiro parâmetro, assim como todo método de instância, é a própria instância. Por convenção, chamamos este argumento de **self**. Vejamos um exemplo:

```
class Conta:
    def __init__(self, numero, titular, saldo, limite):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite
```

Agora, quando uma classe é criada, todos os seus atributos serão inicializados pelo método `__init__()`. Apesar de muitos programadores chamarem este método de construtor, ele não cria um objeto conta. Existe outro método, o `__new__()` que é chamado antes do `__init_()` pelo interpretador do Python. O método `__new__()` é realmente o construtor e é quem realmente cria uma instância de `Conta`. O método `__init__()` é responsável por inicializar o objeto, tanto é que já recebe a própria instância (self) criada pelo construtor como argumento. Dessa maneira, garantimos que toda instância de uma `Conta` tenha os atributos que definimos.

Agora, se executarmos a linha de código abaixo, vai acusar um erro:

```
    conta = Conta()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: __init__() missing 4 required positional arguments: 'numero', 'titular', 'saldo', and 'lim
ite
```

O erro acusa a falta de 4 argumentos na hora de criar uma `Conta`. A classe `Conta` agora nos obriga a passar 4 atributos (numero, titular, saldo e limite) para criar uma conta:

```
conta = Conta('123-4', 'João', 120.0, 1000.0)
```

Veja que em nenhum momento chamamos o método `__init__()`. Quem está fazendo isso por debaixo dos panos é o próprio Python quando executa `conta = Conta()`. Como vimos, ele chama o

método `__new__()` que devolve um instância do objeto e em seguida chama o método `__init__()` toda vez que criamos uma conta. Podemos ver isto funcionando imprimindo uma mensagem dentro do método `__init__()`:

```
def __init__(self, titular, numero, saldo, limite):
    print("inicializando uma conta")
    self.titular = titular
    self.numero = numero
    self.saldo = saldo
    self.limite = limite
```

e testar novamente:

```
    conta = Conta('123-4', 'João', 120.0, 1000.0)
inicializando uma conta
```

Ao criar uma `Conta`, estamos pedindo para o Python criar uma nova instância de `Conta` na memória, ou seja, o Python alocará memória suficiente para guardar todas as informações da `Conta` dentro da memória do programa. O `__new__()`, portanto, devolve uma **referência**, uma seta que aponta para o objeto em memória e é guardada na variável `conta`.

Para manipularmos nosso objeto `conta` e acessarmos seus atributos, utilizamos o operador "." (ponto):

```
print(conta.titular)
#'João'
print(conta.saldo)
#120.0
```

Como o **self** é a referência do objeto, ele chama `self.titular` e `self.saldo` da classe `Conta`.

Agora, além de funcionar como esperado, nosso código não permite criar uma conta sem os atributos que definimos anteriormente. Discuta com seus colegas e instrutor as vantagens da orientação a objetos até aqui.

Mais adiante, veremos que algumas vezes é mais interessante lançar uma exceção (*exception*) nesses casos.

8.7 OBJETOS SÃO ACESSADOS POR REFERÊNCIA

O programa pode manter na memória não apenas uma `Conta`, mas mais de uma:

```
minha_conta = Conta()
minha_conta.saldo = 1000

meu_sonho = Conta()
meu_sonho.saldo = 1500000
```

Quando criamos uma variável para associar a um objeto, na verdade, essa variável não guarda o objeto, e sim uma maneira de acessá-lo, chamada de **referência** (o self).

```
c1 = Conta()
```

Ao fazer isso, já sabemos que o Python está chamando os métodos mágicos `__new__()` e `__init__()` que são responsáveis por construir e iniciar um objeto do tipo `Conta`.

O correto é dizer que `c1` se refere a um objeto. Não é correto dizer que `c1` é um objeto, pois `c1` é uma variável referência. Todavia, é comum ouvir frases como “tenho um objeto `c1` do tipo `Conta`”, mas isso é apenas uma abreviação para encurtar a frase “Tenho uma referência `c1` a um objeto tipo `Conta`”.

Vamos analisar o código abaixo:

```
c1 = Conta('123-4', 'João', 120.0, 1000.0)
c2 = c1
print(c2.saldo)
#120.0
c1.deposita(100.0)
print(c1.saldo)
#220.0
c2.deposita(30.0)
print(c2.saldo)
#250.0
print(c1.saldo)
#250.0
```

O que aconteceu aqui? O operador “=” copia o valor de uma variável. Mas qual é o valor da variável `c1`? É o objeto? Não. Na verdade, o valor guardado é a referência (endereço) de onde o objeto se encontra na memória principal.

```
print(minha_conta.titular)
#<__main__.Cliente object at 0x7f83dac31dd8>
```

Veja que a saída é a referência a um objeto do tipo `Cliente`, mas podemos acessar seus atributos de uma forma mais direta e até mais elegante:

```
print(minha_conta.titular.nome)
#'João'
```

8.10 TUDO É OBJETO

Python é uma linguagem totalmente orientada a objetos. Tudo em Python é um objeto! Sempre que utilizamos uma função ou método que recebe parâmetros, estamos passando objetos como argumentos. Não é diferente com nossas classes. Quando uma conta recebe um cliente como titular, ele está recebendo uma instância de `Cliente`, ou seja, um objeto.

O mesmo acontece com `numero`, `saldo` e `limite`. *Strings* e números são classes no Python. Por este motivo que aparece a palavra **class** quando pedimos para o Python nos devolver o tipo de uma variável através da função **type**:

```
print(type(conta.numero))
#<class 'str'>
print(type(conta.saldo))
#<class 'float'>
print(type(conta.titular))
#<class '__conta__.Cliente'>
```

Um sistema orientado a objetos é um grande conjunto de classes que vai se comunicar, delegando responsabilidades para quem for mais apto a realizar determinada tarefa. A classe `Banco` usa a classe `Conta`, que usa a classe `Cliente`, que usa a classe `Endereco`, etc... Dizemos que esses objetos colaboram, trocando mensagens entre si. Por isso, acabamos tendo muitas classes em nosso sistema, e elas costumam ter um tamanho relativamente curto.

CAPÍTULO 9

MODIFICADORES DE ACESSO E MÉTODOS DE CLASSE

Um dos problemas mais simples que temos no nosso sistema de contas é que o método `saca()` permite sacar mesmo que o saldo seja insuficiente. A seguir, você pode lembrar como está a classe `Conta`:

```
class Conta:

    def __init__(self, numero, titular, saldo, limite=1000.0):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite

    # outros métodos

    def saca(self, valor):
        self.saldo -= valor
```

Vamos testar nosso código:

```
minha_conta = Conta('123-4', 'João', 1000.0, 2000.0)
minha_conta.saca(500000)
```

O limite de saque é ultrapassado. Podemos incluir um `if` dentro do método `saca()` para evitar a situação que resultaria em uma conta em estado inconsistente, com seu saldo menor do que zero. Fizemos isso no capítulo de orientação a objetos básica.

Apesar de melhorar bastante, ainda temos um problema mais grave: ninguém garante que o usuário da classe vai sempre utilizar o método para alterar o saldo da conta. O código a seguir altera o saldo diretamente:

```
minha_conta = Conta('123-4', 'João', 1000.0)
minha_conta.saldo = -200
```

Como evitar isso? Uma ideia simples seria testar se não estamos sacando um valor maior que o saldo toda vez que formos alterá-lo.

```
    minha_conta = Conta('123-4', 'joão', 1000.0)
    novo_saldo = -200

if (novo_saldo < 0):
    print("saldo inválido")
else:
    minha_conta.saldo = novo_saldo
```

acessando seus métodos. Algo muito mais educado que mexer diretamente em um atributo que não é seu.

Melhor ainda! O dia que precisarmos mudar como é realizado um saque na nossa classe `Conta`, adivinhe onde precisaríamos modificar? Apenas no método `saca()`, o que faz pleno sentido.

9.1 ENCAPSULAMENTO

O que começamos a ver nesse capítulo é a ideia de encapsular, isto é, 'esconder' todos os membros de uma classe (como vimos acima), além de esconder como funcionam as rotinas (no caso métodos) do nosso sistema.

Encapsular é fundamental para que seu sistema seja suscetível a mudanças: não precisamos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está encapsulada. O conjunto de métodos públicos de uma classe é também chamado de **interface da classe**, pois esta é a única maneira a qual você se comunica com objetos dessa classe.

O *underscore* _ alerta que ninguém deve modificar, nem mesmo ler, o atributo em questão. Com isso, temos um problema: como fazer para mostrar o saldo de uma `Conta`, já que não devemos acessá-lo para leitura diretamente?

Precisamos então arranjar uma maneira de fazer esse acesso. Sempre que precisamos arrumar uma maneira de fazer alguma coisa com um objeto, utilizamos métodos! Vamos então criar um método, digamos `pega_saldo()`, para realizar essa simples tarefa:

```
class Conta:

    # outros métodos

    def pega_saldo(self):
        return self._saldo
```

Para acessarmos o saldo de uma conta, podemos fazer:

```
    minha_conta = Conta('123-4', 'joão', 1000.0)
    minha_conta.deposita(100)
    minha_conta.pega_saldo()
1100
```

Para permitir o acesso aos atributos (já que eles são 'protegidos') de uma maneira controlada, a prática mais comum é criar dois métodos, um que retorna o valor e outro que muda o valor. A convenção para esses métodos em muitas linguagens orientadas a objetos é colocar a palavra **get** ou **set** antes do nome do atributo. Por exemplo, uma conta com saldo e titular fica assim, no caso de desejarmos dar acesso a leitura e escrita a todos os atributos:

```
class Conta:

    def __init__(self, titular, saldo):
```

Capítulo 11

HERANÇA E POLIMORFISMO

11.1 REPETINDO CÓDIGO?

Como toda empresa, nosso banco possui funcionários. Um funcionário tem um nome, um cpf e um salário. Vamos modelar a classe `Funcionario` :

```
class Funcionario:

    def __init__(self, nome, cpf, salario):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario

    # outros métodos e propriedades
```

Além de um funcionário comum, há também outros cargos, como os gerentes. Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes. Um gerente no nosso banco possui também uma senha numérica que permite o acesso ao sistema interno do banco, além do número de funcionários que ele gerencia:

```
class Gerente:

    def __init__(self, nome, cpf, salario, senha, qtd_gerenciados):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario
        self._senha = senha
        self._qtd_gerenciados = qtd_gerenciados

    def autentica(self, senha):
        if self._senha == senha:
            print("acesso permitido")
            return True
        else:
           print("acesso negado")
           return False

    # outros métodos (comuns a um Funcionario)
```

Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente.

Além disso, se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaremos passar por todas as classes de funcionário e adicionar esse atributo. O problema acontece novamente por não centralizarmos as informações principais do funcionário em um único lugar!

Existe um jeito de relacionarmos uma classe de tal maneira que uma delas **herda** tudo que o outra tem. Isto é uma relação de **herança**, uma relação entre classe 'mãe' e classe 'filha'. No nosso caso, gostaríamos de fazer com que `Gerente` tivesse tudo que um `Funcionario` tem, gostaríamos que ela fosse uma extensão de `Funcionario`. Fazemos isso acrescentando a classe mãe entre parenteses junto a classe filha:

```
class Gerente(Funcionario):

    def __init__(self, senha, qtd_funcionarios):
        self._senha = senha
        self._qtd_funcionarios = qtd_funcionarios

    def autentica(self, senha):
        if self._senha == senha:
            print("acesso permitido")
            return True
        else:
            print("acesso negado")
            return False
```

Todo momento que criarmos um objeto do tipo `Gerente` queremos que este objeto também herde os atributos definidos na classe `Funcionario`, pois um **Gerente é um Funcionário**.

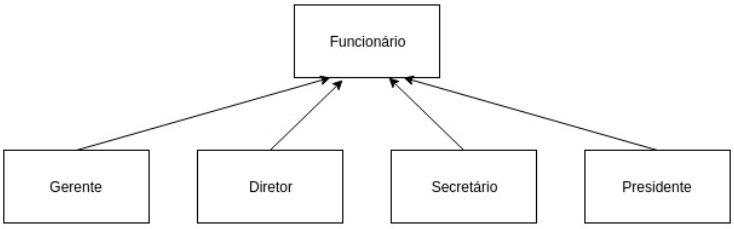
Como a classe `Gerente` já possui um método `__init__()` com outros atributos, o método da classe `Funcionario` é **sobrescrito** pelo `Gerente`. Se queremos incluir os mesmos atributos de instância de `Funcionario` em um `Gerente`, devemos chamar o método `__init__()` de `Funcionario` dentro do método `__init__()` de Gerente:

```
class Gerente(Funcionario):

    def __init__(self, senha, qtd_funcionarios):
        Funcionario.__init__(nome, cpf, salario)
        self._senha = senha
        self._qtd_funcionarios = qtd_funcionarios

    def autentica(self, senha):
        if self._senha == senha:
            print("acesso permitido")
            return True
        else:
            print("acesso negado")
            return False
```

PARA SABER MAIS: SUPER E SUB CLASSE

A nomenclatura mais encontrada é que Funcionario é a **superclasse** de Gerente, e Gerente é a **subclasse** de Funcionario. Dizemos também que todo Gerente **é um** Funcionario. Outra forma é dizer que Funcionario é a classe **mãe** de Gerente e Gerente é a classe **filha** de Funcionario.

Da mesma maneira, podemos ter uma classe `Diretor` que estenda `Gerente`, e a classe `Presidente` pode estender diretamente de `Funcionario`. Fique claro que essa é uma relação de negócio. Se `Diretor` vai estender de `Gerente` ou não, vai depender, para você, `Diretor` é um `Gerente`?



11.2 REESCRITA DE MÉTODOS

Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.

`__repr__()` é usado para representar o objeto de maneira técnica, inclusive podendo utilizá-lo como comando válido do Python, como vimos no exemplo da classe `Ponto`.

11.4 PARA SABER MAIS - MÉTODOS MÁGICOS

Os métodos mágicos são úteis, pois permitem que os objetos de nossas classes possuam uma interface de acesso semelhante aos objetos embutidos do Python. O método `__add__()`, por exemplo, serve para executar a adição de dois objetos e é chamada sempre quando fazemos a operação de adição (obj + obj) utilizando o operador '+'. Por exemplo, quando fazemos `1 + 1` no Python, o que o interpretador faz é chamar o método `__add__()` da classe `int`. Vimos que uma `list` também implementa o método `__add__()`, já que a operação de adição é definida para esta classe:

```
>>> lista = [1, 2, 3]
>>> lista + [4, 5]
[1, 2, 3, 4, 5]
```

O mesmo ocorre para as operações de multiplicação, divisão, módulo e potência que são definidas pelos métodos mágicos `__mul__()`, `__div__()`, `__mod__()` e `__pow__()`, respectivamente.

Podemos definir cada uma dessas operações em nossas classes sobrescrevendo tais métodos mágicos. Além desses, o Python possui muitos outros que você pode acessar aqui: https://docs.python.org/3/reference/datamodel.html#Basic_customization

11.5 POLIMORFISMO

O que guarda uma variável do tipo `Funcionario` ? Uma referência para um `Funcionario` , nunca o objeto em si.

Na herança, vimos que todo `Gerente` é um `Funcionario` , pois é uma extensão deste. Podemos nos referir a um `Gerente` como sendo um `Funcionario` . Se alguém precisa falar com um `Funcionario` do banco, pode falar com um `Gerente` ! Porque? Pois `Gerente` **é um** `Funcionario` . Essa é a semântica da herança.

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

A situação que costuma aparecer é a que temos um método que recebe um argumento do tipo `Funcionario` :

```
class ControleDeBonificacoes:

    def __init__(self, total_bonificacoes=0):
        self._total_bonificacoes = total_bonificacoes

    def registra(self, funcionario):
        self._total_bonificacoes += funcionario.get_bonificacao()

    @property
    def total_bonificacoes(self):
        return self._total_bonificacoes
```

E podemos fazer:

```
if __name__ == '__main__':
    funcionario = Funcionario('João', '111111111-11', 2000.0)
    print("bonificacao funcionario: {}".format(funcionario.get_bonificacao()))

    gerente = Gerente("José", "222222222-22", 5000.0 '1234', 0)
    print("bonificacao gerente: {}".format(gerente.get_bonificacao()))

    controle = ControleDeBonificacoes()
    controle.registra(funcionario)
    controle.registra(gerente)

    print("total: {}".format(controle.total_bonificacoes))
```

que gera a saída:

```
bonificacao funcionario: 200.0
bonificacao gerente: 1500.0
total: 1700.0
```

Repare que conseguimos passar um `Gerente` para um método que "recebe" um `Funcionario` como argumento. Pense como numa porta na agência bancária com o seguinte aviso: "Permitida a entrada apenas de Funcionários". Um gerente pode passar nessa porta? Sim, pois `Gerente` **é um** `Funcionario` .

11.6 DUCK TYPING

Uma característica de linguagens dinâmicas como Python é a chamada **Duck Typing**, a tipagem de pato. É uma característica de um sistema de tipos em que a semântica de uma classe é determinada pela sua capacidade de responder a alguma mensagem, ou seja, responder a determinado atributo (ou método). O exemplo canônico (e a razão do nome) é o teste do pato: *se ele se parece com um pato, nada como um pato e grasna como um pato, então provavelmente é um pato*.

Veja o exemplo abaixo:

```
class Pato:
    def grasna(self):
        print('quack!')

class Ganso:
    def grasna(self):
        print('quack!')

if __name__ == '__main__':
    pato = Pato()
    print(pato.grasna())

    ganso = Ganso()
    print(ganso.grasna())
```

Que gera a saída:

```
quack!
quack!
```

Você deve escrever o código esperando somente uma interface do objeto, não um tipo de objeto. No caso da nossa classe `ControleDeBonificacoes`, o método `registra()` espera um objeto que possua o método `get_bonificacao()` e não apenas um funcionário.

O *Duck Typing* é um estilo de programação que não procura o tipo do objeto para determinar se ele tem a interface correta. Ao invés disso, o método ou atributo é simplesmente chamado ou usado ('se parece como um pato e grasna como um pato, então deve ser um pato'). *Duck Typing* evita testes usando as funções `type()`, `isinstance()` e até mesmo a `hasattr()` - ao invés disso, deixa o erro estourar na frente do programador.

A maneira Pythônica para garantir a consistência do sistema não é verificar os tipos e atributos de um objeto, mas pressupor a existência do atributo no objeto e tratar uma exceção, caso ocorra, através do comando `try/except`:

```
    try:
        self._total_bonificacoes += obj.get_bonificacao()
    except AttributeError as e:
        print(e)
```

Estamos pedindo ao interpretador para tentar executar a linha de código dentro do comando `try` (tentar). Caso ocorra algum erro, ele vai tratar este erro com o comando `except` e executar algo, como

percorra todas as contas do `Banco` para passá-las como argumento para o método `roda()` do `AtualizadorDeContas`.

4. (opcional) Que maneira poderíamos implementar o método `atualiza()` nas classes `ContaCorrente` e `ContaPoupança` poupando reescrita de código?

5. (opcional) E se criarmos uma classe que não é filha de `Conta` e tentar passar uma instância no método roda de `AtualizadorDeContas`? Com o que aprendemos até aqui, como podemos evitar que erros aconteçam nestes casos?

11.8 CLASSES ABSTRATAS

Vamos recordar nossa classe `Funcionario`:

```
class Funcionario:

    def __init__(self, nome, cpf, salario=0):
        #inicialização dos atributos

    #propriedades e outros métodos

    def get_bonificacao(self):
        return self._salario * 1.2
```

Considere agora nosso `ControleDeBonificacao`:

```
class ControleDeBonificacoes:

    def __init__(self, total_bonificacoes=0):
        self.__total_bonificacoes = total_bonificacoes

    def registra(self, obj):
        if(hasattr(obj, 'get_bonificacao')):
            self.__total_bonificacoes += obj.get_bonificacao()
        else:
```

Capítulo 13

EXCEÇÕES E ERROS

Voltando às contas que criamos no capítulo 8, o que aconteceria ao tentarmos chamar o método `saca()` com um valor fora do limite? O sistema mostraria uma mensagem de erro, mas quem chamou o método `saca()` não saberá que isso aconteceu.

Como avisar àquele que chamou o método de que ele não conseguiu fazer aquilo que deveria?

Os métodos dizem qual o contrato que eles devem seguir. Se, ao tentar chamar o método `sacar()`, ele não consegue fazer o que deveria, ele precisa, ao menos, avisar ao usuário que o saque não foi feito.

Veja no exemplo abaixo: estamos forçando uma `Conta` a ter um valor negativo, isto é, a estar em um estado inconsistente de acordo com a nossa modelagem.

```
conta = Conta('123-4', 'João')
conta.deposita(100.0)
conta.saca(3000.0)

#o método saca funcionou?
```

Em sistemas de verdade, é muito comum que quem saiba tratar o erro é aquele que chamou o método, e não a própria classe! Portanto, nada mais natural sinalizar que um erro ocorreu.

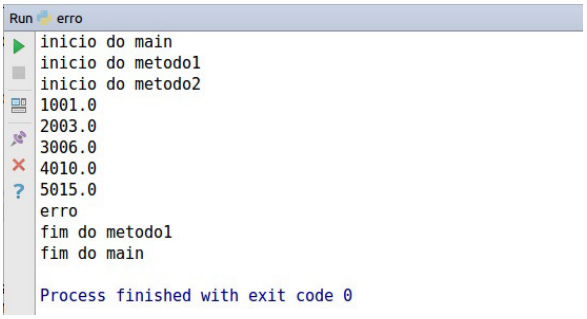
A solução mais simples utilizada antigamente é a de marcar o retorno de um método como boolean e retornar `True` se tudo ocorreu da maneira planejada, ou `False`, caso contrário:

```
if (valor > self.saldo + self.limite):
    print("nao posso sacar fora do limite")
    return False
else:
    self.saldo -= valor
    return True
```

Um novo exemplo de chamada do método acima:

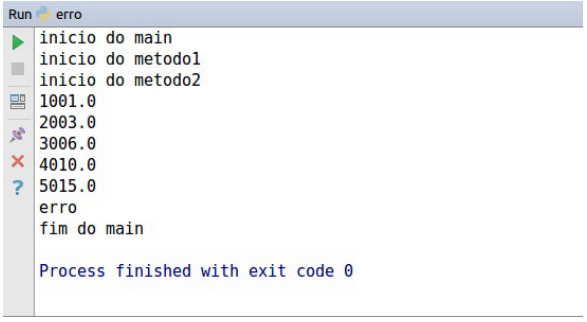
```
conta = Conta('123-4', 'João')
conta.deposita(100.0)
conta.limite = 100.0

if (not conta.saca(3000.0)):
    print("nao saquei")
```

Repare que tivemos de lembrar de testar o retorno do método, mas não somos obrigados a fazer isso. Esquecer de testar o retorno desse método teria consequências drásticas: a máquina de autoatendimento



Faça o mesmo, retirando o `try/except` novamente e colocando-o em volta da chamada do `metodo1()`. Rode os códigos, o que acontece?

```
if __name__ == '__main__':
    print('início do main')
    try:
        metodo1()
    except AttributeError:
        print('erro')
    print('fim do main')
```



Repare que, a partir do momento que uma *exception* foi *catched* (pega, tratada, *handled*), a exceção volta ao normal a partir daquele ponto.

13.1 EXCEÇÕES E TIPOS DE ERROS

Runtime

Este tipo de erro ocorre quando algo de errado acontece durante a execução do programa. A maior parte das mensagens deste tipo de erro inclui informações do que o programa estava fazendo e o local que o erro aconteceu.

O interpretador mostra a famosa *Traceback* - ele mostra a sequência de chamadas de função que fez com que você chegasse onde está, incluindo o número da linha de seu arquivo onde cada chamada ocorreu.

Os erros mais comuns de tempo de execução são:

- **NameError**

Quando tentamos acessar uma variável que não existe.

```
print(x)

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

No exemplo acima, tentamos imprimir `x` sem defini-lo antes. Este erro também é muito comum de ocorrer quando tentamos acessar uma variável local em um contexto global.

- **TypeError**

Quando tentamos usar um valor de forma inadequada, como por exemplo tentar indexar um sequência com algo diferente de um número inteiro ou de um fatiamento:

```
lista = [1, 2, 3]
print(lista['a'])

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: list indices must be integers or slices, not str
```

- **KeyError**

Quando tentamos acessar um elemento de um dicionário usando uma chave que não existe.

```
dicionario = {'nome': 'João', 'idade': 25}
print(dicionario['cidade'])

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'cidade'
```

- **AttributeError**

Quando tentamos acessar um atributo ou método que não existe em um objeto.

```
lista = [1, 2, 3]
print(lista.nome)

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AttributeError: 'list' object has no attribute 'nome'
```

- **IndexError**

Quando tentamos acessar um elemento de uma sequência com um índice maior que o total de seus elementos menos um.

```
tupla = (1, 2, 3)
print(tupla[3])

Traceback (most recent call last):
```

```
  File "<stdin>", line 2, in <module>
IndexError: tuple index out of range
```


13.2 TRATANDO EXCEÇÕES

Há muitos outros erros de tempo de execução. Que tal dividir um número por zero? Será que o interpretador consegue fazer aquilo que nós definimos que não existe?

```
n = 2
n = n / 0

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

Repare que um `ZeroDivisionError` poderia ser facilmente evitado com um `if` que checaria se o denominador é diferente de zero, mas a forma correta de se tratar um erro no Python é através do comando **try/except**:

```
try:
    n = n/0
except ZeroDivisionError:
    print('divisão por zero')
```

Que gera a saída:

```
divisão por zero
```

O conjunto de instruções dentro do bloco `try` é executado (o interpretador tentará executar). Se nenhuma exceção ocorrer, o comando `except` é ignorado e a execução é finalizada. Todavia, se ocorrer alguma exceção durante a execução do bloco `try`, as instruções remanescentes serão ignoradas, e se a exceção lançada prever um `except`, as instruções dentro do bloco `except` serão executadas.

O comando `try` pode ter mais de um comando `except` para especificar múltiplos tratadores para

diferentes exceções. No máximo um único tratador será ativado. Tratadores só são sensíveis às exceções levantadas no interior da cláusula `try`, e não as que tenham ocorrido no interior de outro tratador numa mesma instrução `try`. Um tratador pode ser sensível a múltiplas exceções, desde que as especifique em uma tupla:

```
except(RuntimeError, TypeError, NameError):
    pass
```

A última cláusula `except` pode omitir o nome da exceção, funcionando como um coringa. Não é aconselhável abusar deste recurso, já que isso pode esconder erros do programador e do usuário.

O bloco `try/except` possui um comando opcional `else` que, quando usado, deve ser colocado depois de todos os comandos `except`. É um comando útil para códigos que precisam ser executados se nenhuma exceção foi lançada, por exemplo:

```
try:
    arquivo = open('palavras.txt', 'r')
except IOError:
    print('não foi possível abrir o arquivo')
else:
    print('o arquivo tem {} palavras'.format(len(arquivo.readlines())))
    arquivo.close()
```

13.3 LEVANTANDO EXCEÇÕES

O comando `raise` nos permite forçar a ocorrência de um determinado tipo de exceção. Por exemplo:

```
raise NameError('oi')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: oi
```

O argumento de `raise` indica a exceção a ser lançada. Esse argumento deve ser uma instância de `Exception` ou uma classe de alguma exceção - uma classe que deriva de `Exception`.

Caso você precise determinar se uma exceção foi lançada ou não, mas não quer manipular o erro, uma forma é lançá-la novamente através da instrução `raise`:

```
try:
    raise NameError('oi')
except NameError:
    print('lançou uma exceção')
    raise
```

Saída:

```
lançou uma exceção
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: oi
```

CAPÍTULO 16

APÊNDICE - INSTALAÇÃO

O Python já vem instalado nos sistemas Linux e Mac OS mas será necessário fazer o download da última versão (Python 3.6) para acompanhar a apostila. O Python não vem instalado por padrão no Windows e o download deverá ser feito no site https://www.python.org/ além de algumas configurações extras.

16.1 INSTALANDO O PYTHON NO WINDOWS

O primeiro passo é acessar o site do Python: https://www.python.org/. Na sessão de `Downloads` já será disponibilizado o instalador específico do Windows automaticamente, portanto é só baixar o Python3, na sua versão mais atual.

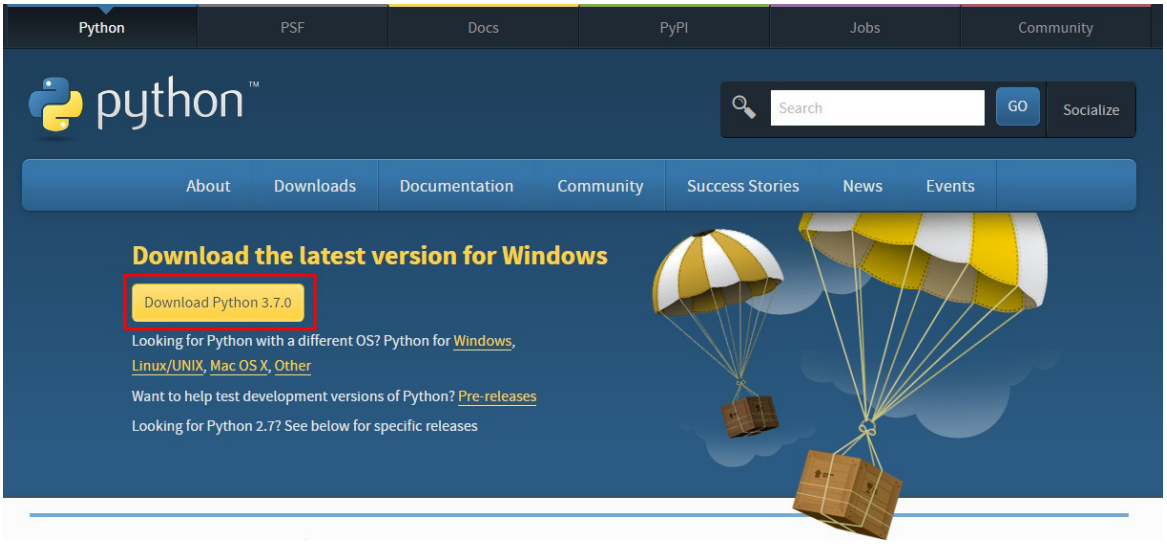


Figura 16.1: Tela de download do Python para o Windows

Após o download ser finalizado, abra-o e na primeira tela marque a opção `Add Python 3.X to PATH`. Essa opção é importante para conseguirmos executar o Python dentro do Prompt de Comando do Windows. Caso você não tenha marcado esta opção, terá que configurar a variável de ambiente no Windows de forma manual.

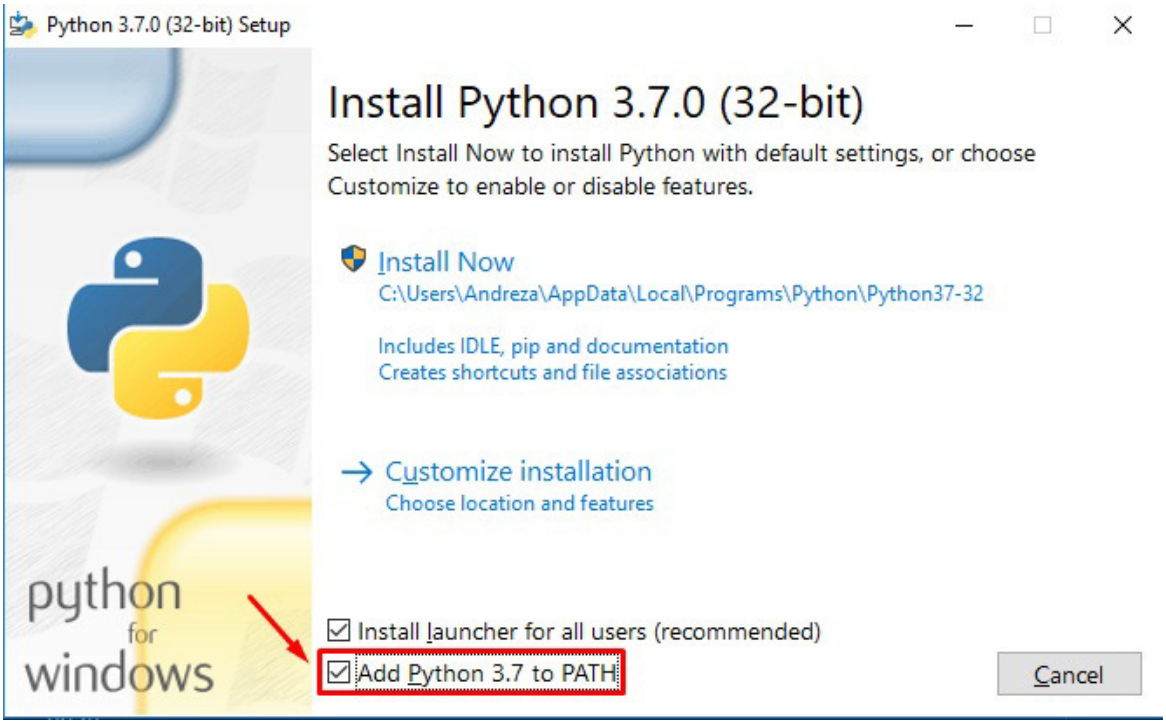


Figura 16.2: Checkbox selecionado

Selecione a instalação customizada somente para ver a instalação com mais detalhes.

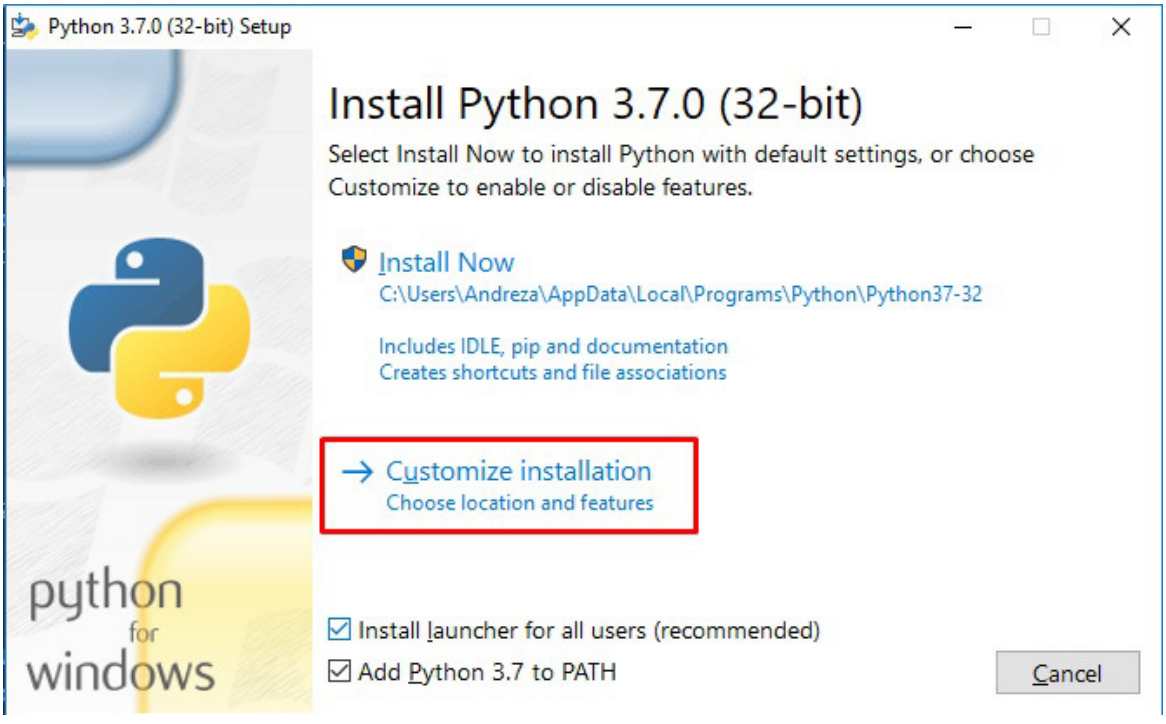


Figura 16.3: Instalação customizada

Na tela seguinte são as features opcionais, se certifique que o gerenciador de pacotes `pip` esteja selecionado, ele que permite instalar pacotes e bibliotecas no Python. Clique em `Next` para dar

seguimento na instalação.

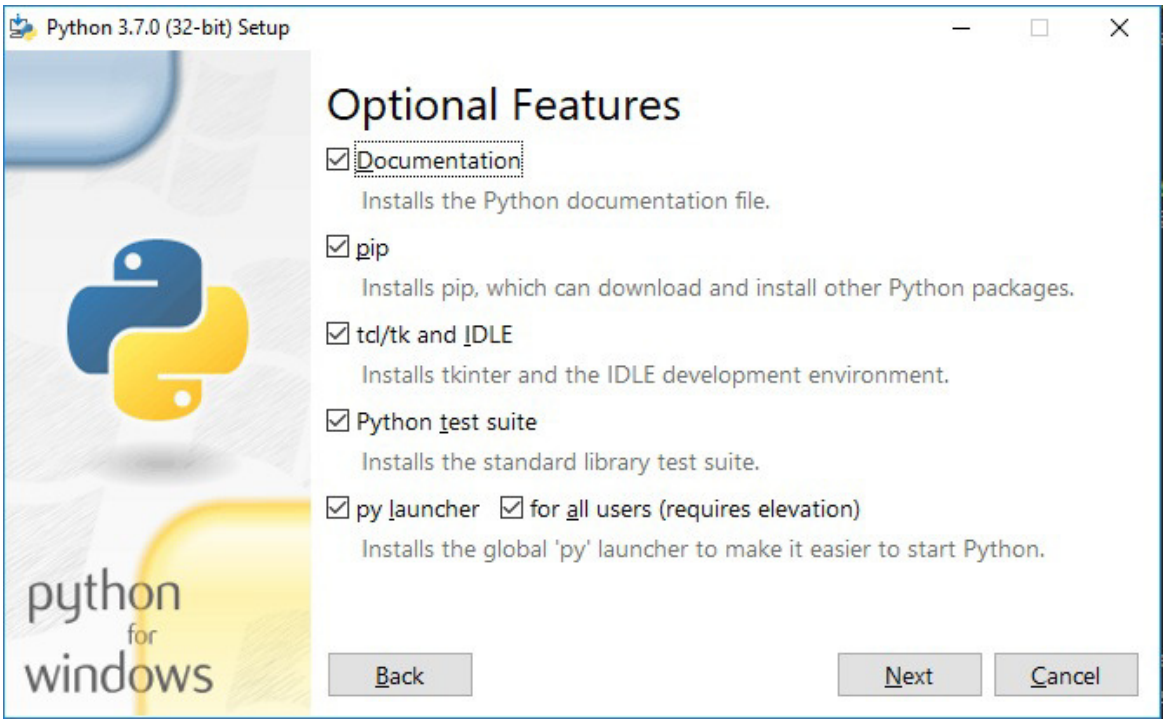


Figura 16.4: Optional Features

Já na terceira tela, deixe tudo como está, mas se atente ao diretório de instalação do Python, para caso queira procurar o executável ou algo que envolva o seu diretório.

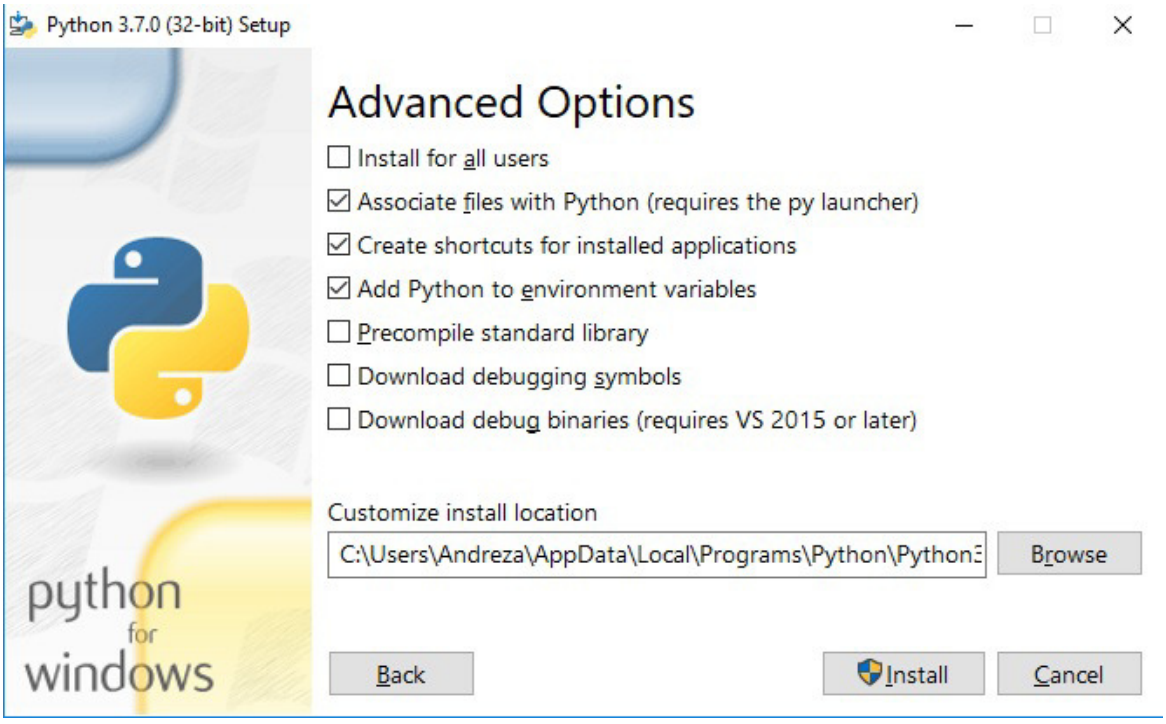


Figura 16.5: Advanced Options

Você pode também fazer o curso data dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso data** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*

16.2 INSTALANDO O PYTHON NO LINUX

Os sistemas operacionais baseados no Debian já possuem o Python3 pré-instalado. Verifique se o seu sistema já possui o Python3 instalado executando o seguinte comando no terminal:

```
python3 -V
```

OBS: O comando `python3 -V` é importante que esteja com o `V` com letra maiúscula.

Este comando retorna a versão do Python3 instalada. Se você ainda não tiver ele instalado, digite os seguintes comandos no terminal:

```
sudo apt-get update
sudo apt-get install python3
```

16.3 INSTALANDO O PYTHON NO MACOS

A maneira mais fácil de instalar o Python3 no MacOS é utilizando o `Homebrew`. Com o `Homebrew` instalado, abra o terminal e digite os seguintes comandos:

```
brew update
brew install python3
```

16.4 OUTRAS FORMAS DE UTILIZAR O PYTHON

Podemos rodar o Python diretamente do seu próprio Prompt.

Podemos procurar pelo Python na caixa de pesquisa do Windows e abri-lo, assim o seu console próprio será aberto. Uma outra forma é abrir a IDLE do Python, que se parece muito com o console mas vem com um menu que possui algumas opções extras.