



Curso Python 3

Autor: Juracy Filho
Co-autor: Leonardo Leitão

Índice

1. Conceitos básicos	2
1.1. O que é Python... uma cobra 🐍?	2
1.2. Mas, morde ?	2
1.3. The Zen of Python	2
2. Python <i>everywhere</i> 🌎	4
2.1. Implementações	4
2.2. Python 2? No thanks! 💔	4
2.3. Instalação	4
3. Interpretador Python	6
3.1. O que é?	6
3.2. Chamando o interpretador	6
3.3. Uso do interpretador	6
4. Ofidioglossia 🐍	8
4.1. Saída padrão - <code>print()</code>	8
4.2. Váriaveis e tipos de dados	8
4.3. <code>__builtins__</code>	10
4.4. Conversão de tipos e coerções	13
4.5. Números	16
4.6. Strings	17
4.7. Listas	19
4.8. Tuplas	22
4.9. Dicionários	23
5. Ninho de cobras	26
5.1. Primeiro módulo	26
5.2. <i>Encoding</i>	26
5.3. <i>Shebang</i>	27
5.4. Baterias inclusas	27
5.5. Entrada de dados	28
5.6. Nome do módulo	29
5.7. E se	29
5.8. Quebrando nosso código em funções	30
5.9. Funções podem retornar valores	31
5.10. Passando argumentos pela linha de comando	32
5.11. Verificação dos argumentos	32
5.12. Melhorando um pouco o nosso <i>help</i>	33
5.13. Dando retorno ao sistema operacional	34
5.14. Validando argumentos	35
6. Instruções	38

6.1. Conhecendo o <code>while</code>	38
6.2. Execute enquanto	38
6.3. <i>Packing</i>	39
6.4. Conhecendo o <code>for</code>	39
6.5. Totalizando uma lista	40
6.6. Forçando a quebra de um laço	40
6.7. Gerando uma lista de números	41
6.8. Recursão	42
6.9. Operador ternário ... <code>if</code> ... <code>else</code>	43
7. Manipulação de arquivos	46
8. Comprehension	50
9. Programação funcional	52
9.1. Capacidades implementadas	52
9.2. <i>First Class Functions</i> - Funções de primeira classe	52
9.3. <i>High Order Functions</i> - Funções de alta ordem	53
9.4. <i>Closure</i> - Funções com escopos aninhados	54
9.5. <i>Anonymous Functions</i> - Funções anônimas (<code>lambda</code>)	54
9.6. Recursion - Recursividade	56
9.7. <i>Immutability</i> - Imutabilidade	57
9.8. <i>Lazy Evaluation</i> - Avaliação preguiçosa	60
9.9. Nem tudo são flores... ☺	63
10. Às funções e além!	66
10.1. Tipos de parâmetros	66
10.2. Parâmetros nomeados ou opcionais	66
10.3. Argumentos nomeados	67
10.4. <i>Unpacking</i> de argumentos	68
10.5. Combinando <i>unpacking</i> e parâmetros opcionais	69
10.6. <i>Unpacking</i> de argumentos nomeados	69
10.7. Objetos chamáveis	70
10.8. Problemas com argumentos mutáveis	71
10.9. Decorators	72
11. Dominando as instruções Python	76
11.1. E se... senão se...	76
11.2. <i>Switch? Case?</i> Não, obrigado!	76
11.3. Laços condicionais <i>like a boss</i>	78
11.4. Iterações <i>like a boss</i>	79
11.5. Tratamento de exceções <i>like a boss</i>	81
12. Packages	83
13. Programação orientada a objetos	86
13.1. Classe Task	86
13.2. Classe Project	86

13.3. Método <code>__iter__</code>	87
13.4. Implementação do vencimento	88
13.5. Herança	89
13.6. Métodos “privados”	90
13.7. Sobrecarga de operador	90
13.8. <i>Snake trap</i>	91
14. Orientada a objetos - Avançado	95
14.1. Membros de classe × membros da instância	95
14.2. Métodos em profundidade	95
14.3. Propriedades	97
14.4. Classe abstrata	99
14.5. Herança Múltipla	101
14.6. <i>Mixins</i>	103
14.7. Protocolo <i>Iterator</i>	105
Anexo A: Soluções	108
Área do Quadrado	108
Fibonacci	108
Manipulação de arquivos	109
Tabuada com <i>List Comprehension</i>	109
MDC	110
Gerador de HTML	110
Palavras proibidas com <code>set</code>	110
Criação de um pacote	111
Controle de vendas de uma loja	111
Contador de objetos	113
Anexo B: Exemplos avançados	114
Fibonacci	114
Fibonacci com <i>memoize</i>	114
Tratamento de CSV com <i>download</i>	115
MDC	116
Várias soluções para fatorial	117
Solução recursiva para a Torre de Hanoi	117
Anexo C: Listas auxiliares	119
Lista de tabelas	119
Lista de figuras e diagramas	119
Anexo D: Listagem de Códigos	120
Exercícios	120
Soluções de desafios	123
Exemplos avançados	123
Glossário	124

Sumário

Apostila do curso de Python.

1. Conceitos básicos

1.1. O que é Python... uma cobra &?

Python é uma linguagem de programação de alto nível, interpretada, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte. Foi lançada por **Guido van Rossum** em 1991. Atualmente possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos **Python Software Foundation**. Apesar de várias partes da linguagem possuírem padrões e especificações formais, a linguagem como um todo não é formalmente especificada. O padrão de *facto* é a implementação **CPython**.

A linguagem foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Prioriza a legibilidade do código sobre a velocidade ou expressividade. Combina uma sintaxe concisa e clara com os recursos poderosos de sua biblioteca padrão e por módulos e frameworks desenvolvidos por terceiros.

Python é uma linguagem de propósito geral de alto nível, multi paradigma, suporta o paradigma orientado a objetos, imperativo, funcional e procedural. Possui tipagem dinâmica e uma de suas principais características é permitir a fácil leitura do código e exigir poucas linhas de código se comparado ao mesmo programa em outras linguagens.

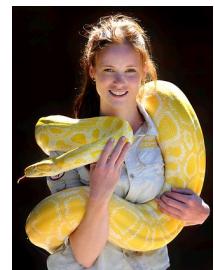
O nome Python teve a sua origem no grupo humorístico britânico **Monty Python**, criador do programa **Monty Python's Flying Circus**, embora muitas pessoas façam associação com o réptil do mesmo nome.

Posteriormente a cobra começou a ser adotada como logo da linguagem.

Referência: [Wikipedia](#)

1.2. Mas, morde ?

Comparada com outras linguagens de mercado, Python tem se sobressaído pela simplicidade, já sendo adotado por diversas universidades pelo mundo como primeira linguagem em diversos cursos de **Tecnologia da Informação**.



Python é provavelmente a linguagem mais usada no mundo por não programadores, tanto que é extremamente comum palestras ministradas por cientistas nas convenções, como: biólogos, matemáticos, físicos, bioquímicos, engenheiros, etc.

Outro fato relevante sobre esta simplicidade se dá pela sua filosofia básica: **The Zen of Python**.

1.3. The Zen of Python

Tim Peters escreveu uma espécie de poema sobre os conceitos da linguagem, que acabou se tornando parte da especificação da mesma na [PEP 20 — The Zen of Python](#).

Este poema se encontra disponível nos interpretadores através do importação do módulo `this`:
`import this.`

A integra do poema

- *Beautiful is better than ugly.*
- *Explicit is better than implicit.*
- *Simple is better than complex.*
- *Complex is better than complicated.*
- *Flat is better than nested.*
- *Sparse is better than dense.*
- *Readability counts.*
- *Special cases aren't special enough to break the rules.*
- *Although practicality beats purity.*
- *Errors should never pass silently.*
- *Unless explicitly silenced.*
- *In the face of ambiguity, refuse the temptation to guess.*
- *There should be one-- and preferably only one --obvious way to do it.*
- *Although that way may not be obvious at first unless you're Dutch.*
- *Now is better than never.*
- *Although never is often better than **right** now.*
- *If the implementation is hard to explain, it's a bad idea.*
- *If the implementation is easy to explain, it may be a good idea.*
- *Namespaces are one honking great idea — let's do more of those!*



Uma tradução/interpretação livre em quadrinhos pode ser encontrada em
<http://hacktoon.com/log/2015/programming-comics-3>

2. Python *everywhere*

2.1. Implementações

A linguagem **Python** atualmente possui inúmeras implementações, sendo a implementação oficial o **C_Python** que trabalha com uma máquina virtual e compilação em *bytecode*.

Existem ainda inúmeras outras implementação, durante este curso focaremos no CPython que é multiplataforma, mas abaixo seguem algumas outras implementações.

- Pypy — Python em Python, permitindo diversas transpilações como em C
- IronPython — .Net
- Jython — Java
- RPython
- Transcript - Rodar Python no *browser* através de transpilação para javascript

2.2. Python 2? No thanks!

A versão atual do Python no momento da escrita deste material é **3.6.4**, em toda a série 3.x tivemos poucas mudanças com potencial de quebrar algo já produzido, porém houve uma quebra bastante significativa da versão 2.x para a 3.x. Em 2009 a versão **3.0** foi lançada trazendo a unificação dos tipos *string* e *unicode*, essa mudança era extremamente necessária para tornar a linguagem mais simples e resolver em definitivo diversos problemas de internacionalização, porém era capaz de quebrar muito código já existente.

A partir deste momento apenas mais uma nova versão da série 2 (sem contar os *fixes*) foi lançada, a versão **2.7.0** em 2010, com o objetivo de aproximar um pouco mais do **Python 3** e servir como plataforma de migração para a nova série.

Então o **Python 2** foi congelado e só recebeu correções, a última foi lançada em 2017: **2.7.14**.

A migração dos sistemas existentes para o **Python 3** demorou mais do que o esperado, principalmente pela baixa adesão inicial de bibliotecas mais utilizadas, porém atualmente não há dúvidas, se vai começar algo novo, **Python 3** por favor!

2.3. Instalação

A maioria das distribuições Linux já trazem consigo o CPython, bastante fácil de verificar chamando na linha de comando: `python --version`.

```
$ python --version
Python 3.6.4
```

O curso é todo focado nas versões mais recentes do **Python 3**, caso o resultado seja 2.x ou alguma versão inferior ao 3.4, sugerimos uma atualização.



Em algumas distribuições Linux optou-se por deixar a versão 2 como python e ter um segundo comando para a versão 3.x do Python, chamado python3.

Caso possua uma versão muito antiga ou não tenha o Python instalado, baixe-o através do link: <https://www.python.org/downloads> ou junto com o fornecedor do seu sistema operacional.

3. Interpretador Python

3.1. O que é?

O **C**Python também possui um interpretador interativo, o que permite experimentos mais imediatos e é bastante útil no aprendizado. Ainda é possível o uso de um interpretador ainda mais amigável chamado **ipython**.



Usaremos fortemente o interpretador padrão do **C**Python durante este curso.

3.2. Chamando o interpretador

```
$ python
Python 3.6.4 (default, Jan  5 2018, 02:35:40)
[GCC 7.2.1 20171224] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



Para fins de simplificação vamos mostrar o código do exemplos independente se rodado a partir de um módulo (arquivo com código Python) ou do interpretador.

3.3. Uso do interpretador

Para que possamos experimentar um pouco o interpretador vamos adiantar alguns assuntos que serão desatrancados melhor em capítulos posteriores. Ao digitar uma expressão no interpretador a mesma é executada e logo após o retorno da expressão é apresentado, a exceção é quando esta expressão for `None`, o equivalente a `null` em Python.

Exemplo

```
>>> 2+2
4
>>>
```

Algumas expressões matemáticas básicas para experimentarmos no interpretador:

- `2+2`
- `5-3`
- `2*3`
- `10/3`
- `10//3`
- `3**2`
- `10%3`
- `None`

A precedência dos operadores pode ser ajustada através de parenteses. A lista completa pode ser encontrada em [Operator precedence](#).

- $2 * 3 + 1 \Rightarrow 7$
- $2 * (3 + 1) \Rightarrow 8$

O resultado da expressão anterior é guardado em uma variável temporária chamada `_`:

- `3*3`
- `_+1`

4. Ofidioglossia &

Vamos começar agora a explorar realmente a linguagem Python.

4.1. Saída padrão - `print()`

Vimos anteriormente que durante o uso do interpretador qualquer expressão terá o seu resultado imediatamente impresso, porém em um módulo Python isso não ocorre. Tudo que precisamos enviar para a saída padrão (normalmente um *console*) precisa ser explicitado, e para isso temos a função `print()`.

O `print()` sempre retorna `None`, o faz que o interpretador não emitirá nada, porém o próprio `print()` enviará para a saída padrão o resultado da expressão passada para ele.

Exercício 1 - Alô Mundo

`alo_mundo.py`

```
>>> print('alo mundo')
alo mundo
```



Este e todos os exercícios estarão disponíveis para download, e você poderá executar este simplesmente chamando: `python alo_mundo.py`.

4.2. Váriaveis e tipos de dados

Como na maioria das linguagens temos o conceito de variáveis e tipos de dados e apesar da linguagem ser tipada dinamicamente, ela é fortemente tipada como veremos em breve.

Já vimos diversos operadores matemáticos, e agora veremos o operador de atribuição `=`, e veremos nossa primeira função nativa do Python: `print()`.

Exercício 2 - Atribuição

`atribuicao.py`

```
>>> a = 10
>>> b = 5
>>> print(a + b)
15
```

Até o momento conhecemos basicamente três tipos de dados: inteiros (`int`), ponto flutuante (`float`) e string (`str`).

Tabela 1. Tipos básicos de dados

Tipo	Seção	Exemplos
bool		True ou False
int	Números	3
float	Números	3.3
str	Strings	'João da Silva' ou "João da Silva"
list	Listas	[1, 2, 'ab']
dict	Dicionários	{'nome': 'João da Silva', 'idade': 21}
NoneType		None

Também conhecemos diversos operadores, como listados na tabela abaixo.

Tabela 2. Operadores

Símbolo	Descrição
+	Soma ou Concatenação
-	Subtração
*	Multiplicação
/	Divisão
//	Divisão de inteiros
**	Exponenciação
%	Módulo da divisão de inteiros
=	Atribuição, coloca o resultado da expressão a direita na identificação (variável) a esquerda



Mesmo os tipos básicos em Python são implementados através de classes. E podem possuir métodos.



Devido ao suporte de sobrecarga de operadores, estes operadores podem ter funções diferentes dependendo das classes dos objetos na expressão.

Os números inteiros também podem ser expressos em diversas bases numéricas diferentes:

Tabela 3. Literais para inteiros em outras bases numéricas

Base numérica	Exemplo	Descrição
Binário	0b111	7 em base decimal
Octal	0o1	8 em base decimal
Hexadecimal	0xff	255 em base decimal



A especificação completa pode ser encontrada em [PEP 3127—Integer Literal Support and Syntax](#).

4.3. __builtins__

Na seção anterior aprendemos um pouco a respeito da função `print()`, agora vamos explicar um pouco por que ela foi apresentada como nativa.

Em python todos os símbolos (*variáveis, classes, funções, etc*) necessários precisam ser importados para estarem disponíveis, e até agora ainda não vimos como fazer isso, porém existe um módulos "embutido" da linguagem chamada `__builtins__`, que é importado automaticamente, é neste módulo que os tipos de dados mais básicos são definidos, e um grande conjunto de funções estão automaticamente disponíveis.



Existe um convenção no python sobre identificadores (*nomes de variáveis, classes, métodos, ...*) circundados por dois *underscores*, se referem a identificadores especiais, normalmente providos pelo próprio Python.

Iremos usar o interpretador agora para inspecionar melhor este módulo e conhecer melhor a linguagem.

Vamos começar pela função `dir()`, com ela podemos listar todos os membros do escopo atual (sem parâmetros) ou de um determinado objeto.

```
dir()
#['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']

pi = 3.1415
dir()
#['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'pi']
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError',
'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FileExistsError', 'FileNotFoundException', 'FloatingPointError', 'FutureWarning', 'GeneratorExit',
'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError',
'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning',
'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError',
'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit',
'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError',
'__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float',
'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

A partir de agora, vamos aprender a utilizar a linguagem para nos ajudar a entendê-la.

Como vimos o `__builtins__` tem mais de 150 membros, não é produtivo conhecê-los todos agora, por isso vamos focar nos mais importantes, principalmente os que nos ajuda a entender melhor a linguagem.

Função `help()`, sem nenhum parâmetro faz o interpretador entra em modo de *help*, o que nos mostrará qualquer ajuda relacionado ao que for digitado. Porém a receber um parâmetro a função `help()` nos mostrar o *help* associado ao objeto.

```
>>> help(dir)
Help on built-in function dir in module builtins:

dir(...)
    dir([object]) -> list of strings

    If called without an argument, return the names in the current scope.
    Else, return an alphabetized list of names comprising (some of) the attributes
    of the given object, and of attributes reachable from it.
    If the object supplies a method named __dir__, it will be used; otherwise
    the default dir() logic is used and returns:
        for a module object: the module's attributes.
        for a class object: its attributes, and recursively the attributes
        of its bases.
        for any other object: its attributes, its class's attributes, and
        recursively the attributes of its class's base classes.
```

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep: string inserted between values, default a space.
    end: string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Função `type()`, nossa primeira função que tem parâmetros obrigatórios. Ela irá retornar o tipo/classe a que pertence o objeto usado como parâmetro.

Exercício 3 - Função `type()`

`type.py`

```
>>> type() ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: type() takes 1 or 3 arguments
>>> type(1) ②
<class 'int'>
>>> type('Alo Mundo')
<class 'str'>
>>> type(10/3)
<class 'float'>
>>> nome = 'João da Silva'
>>> type(nome)
<class 'str'>
>>>
```

- ① A função `type()` exige 1 ou 3 parâmetros, e como não informamos nenhum, o Python levanta uma exceção do tipo `TypeError`. Apesar do nome da exceção remeter ao da função neste caso, é apenas uma coincidência, esta exceção ocorre sempre que parâmetros obrigatórios não são informados por exemplo
- ② Cabe observar que usando o interpretador o retorno das expressões são automaticamente impressas no console, tornando desnecessário o uso do `print()`



O uso do `print()` continua sendo necessário durante a execução de programas

4.4. Conversão de tipos e coerções

Apesar da linguagem possuir o recurso de tipagem dinâmica, ela também é fortemente tipada, conceitos que normalmente não andam juntos.

O que significa que operações normalmente precisam lidar objetos de mesma classe/tipo, para alguns casos existem conversões automáticas ou implícitas (*coerção*) e em outros é necessário converter explicitamente estes objetos.



Classes e tipos são a mesma coisa, pois não existe o conceito de tipos primitivos.

4.4.1. Conversão de tipos

Por exemplo, a soma de um inteiro com uma *string*, mesmo que o conteúdo da *string* seja um número, exige conversão explícita.

Exercício 4 - TypeError

type_error.py

```
>>> print(2 + '2')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

A execução do código acima provoca uma erro (*exceção*) do tipo `TypeError` (*membro do __builtins__*), que indica que ocorreu um erro de tipo. A mensagem ainda deixa claro que a execução do operador `+` de um inteiro (`int`) com uma *string* (`str`) não é suportada.

O que ocorre neste caso é que o resultado esperado da operação não está claro, deveria ser um 4 (*soma*) ou 22 (*concatenação*)?



Basicamente vemos aqui a implementação de um dos conceitos do *The Zen of Python*.

In the face of ambiguity, refuse the temptation to guess ⇒ **Diante da ambiguidade, negue a tentação de adivinhar!**

A solução correta seria converter um dos objetos em outro tipo e executar a soma (*ou concatenação, no caso de strings*), como vemos no exemplo abaixo:

Exercício 5 - Conversão de tipos

soma_int_str.py

```
>>> a = 2
>>> b = '2'
>>> type(a)
<class 'int'>
>>> type(b)
<class 'str'>
>>> a + int(b) ①
4
>>> str(a) + b ②
'22'
>>> type(str(a))
<class 'str'>
>>>
```

① Chamar a classe como uma função cria um novo objeto deste tipo, o que pode ser usado muitas vezes para simplesmente executar uma conversão de dados, passando o valor original como parâmetro

② O operador + teve um comportamento diferente dependendo dos tipos, soma ou concatenação

4.4.2. Coerção (*coercion*)

Existem também situações em que uma operação com tipos diferentes tem um óbvio resultado esperado e nestes casos uma coerção será aplicada automaticamente, como no próximo exemplo.

Exercício 6 - Coerção de tipos

coercaoAutomatica.py

```
>>> 10 / 2  ①
5.0
>>> type(10 / 2)
<class 'float'>
>>> 10 / 3
3.333333333333335
>>> 10 // 3  ②
3
>>> type(10 // 3)
<class 'int'>
>>> 10 / 2.5
4.0
>>> 2 + True  ③
3
>>> 2 + False
2
>>> type(1 + 2)  ④
<class 'int'>
>>> type(1 + 2.5)
<class 'float'>
```

- ① A partir da versão 3, a divisão mesmo entre números inteiros sempre retorna um float (*ponto flutuante*)
- ② O operador // realiza a divisão de números e sempre retorna um int, truncando se necessário
- ③ Em operações numéricas, objetos do tipo bool, retornam 1 para True e 0 para False
- ④ Fora a divisão, outras operações entre dois números podem retornar int ou float, conforme o caso



Uma coisa interessante da coerção de tipo é que tudo pode ser interpretado como um valor booleano, isso permite diversas construções lógicas simples. Veremos isso em detalhes mais adiante.

4.5. Números

Existem diversos tipos para lidar com números, vamos focar nos mais comuns que já estão disponíveis diretamente no `__builtins__`: `int` e `float`. Vamos a alguns exemplos.

Exercício 7 - Números

`numeros.py`

```
>>> dir(int) ①
[..., 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
>>> dir(float) ②
[..., 'as_integer_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real']
>>> a = 5
>>> b = 2.5
>>> a / b ③
2.0
>>> a + b
7.5
>>> a * b
12.5
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(a - b)
<class 'float'>
```

- ① Lista dos membros (*métodos, constantes, ...*) disponíveis para o tipo `int`, uma dica é utilizar a função `help(int)` que detalhará melhor cada um deles
- ② Lista dos membros (*métodos, constantes, ...*) disponíveis para o tipo `float`, uma dica é utilizar a função `help(float)` que detalhará melhor cada um deles
- ③ Normalmente operações envolvendo os dois tipos retornam `float`

No `__builtins__` além de `int` e `float`, temos também o `complex`, como indicado na documentação oficial: [Numeric Types](#).



Outro tipo numérico que vale menção é o `Decimal`:

O tipo `Decimal` não faz parte do `__builtins__` e precisa ser importado antes do seu uso

```
from decimal import Decimal
```

4.6. Strings

O tipo `str` serve para lidar com cadeias de texto, e entre os tipos básicos é um dos que mais métodos e recursos possui. Vamos a alguns exemplos.

Exercício 8 - Strings

`strings.py`

```
>>> dir(str) ①
[..., 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> nome = 'Juracy Filho' ②
>>> "Dias D'Avila" == 'Dias D\Avila' ③
>>> texto = 'Texto entre apostrófes pode ter "aspas"' ④
>>> doc = """Texto com múltiplas
... linhas"""\n⑤
>>> doc2 = '''Também é possível
... com aspas simples''' ⑥
>>> nome ⑦
'Juracy Filho'
>>> print(nome)
Juracy Filho
>>> doc
'Texto com múltiplas\nlinhas' ⑧
>>> print(doc)
Texto com múltiplas
linhas
```

- ① Lista dos membros (*métodos, constantes, ...*) disponíveis para o tipo `str`, uma dica é utilizar a função `help(str)` que detalhará melhor cada um deles, este tipo possui um *help* bastante extenso e pode ser interessante consultar o *help* de um método em específico, por exemplo: `help(str.upper)`
- ② Existem diversas formas de expressar uma *string*, neste material vamos priorizar o uso de aspas simples, mas funcionaria normalmente com aspas duplas
- ③ Apesar de ser possível utilizar *backslash* (\) para escapar caracteres, no caso da própria aspa simples é aconselhado delimitar a *string* com aspas duplas
- ④ Da mesma forma que aspas podem ser utilizadas dentro das aspas simples, devemos evitar uso do escape de forma desnecessária
- ⑤ É possível especificar *strings* com múltiplas linhas usando 3 aspas duplas
- ⑥ Apesar de ser possível utilizar 3 aspas simples, a [PEP-8](#) recomenda utilizar aspas duplas
- ⑦ No interpretador a saída automática do resultado da expressão não é exatamente igual ao `print()`, na realidade é a representação do objeto, com *strings* isso fica mais claro
- ⑧ A representação de quebra de linha é feita através da sequência: \n

Exercício 9 - Métodos e operadores para Strings

strings_methods.py

```
>>> nome = 'Juracy Filho'  
>>> nome[:6] ①  
'Juracy'  
>>> 're' in nome ②  
False  
>>> 'ra' in texto  
True  
>>> len(nome) ③  
12  
>>> nome.lower() ④  
'juracy filho'  
>>> nome.upper() ⑤  
'JURACY FILHO'  
>>> nome.split() ⑥  
['Juracy', 'Filho']
```

① *Strings* suportam indexação e fatiamento, nestes casos se comportam como uma lista de caracteres, mas detalhes em [Exercício 11 - Indexação das Listas](#) e [Exercício 12 - Fatiamento de Listas](#)

② O operador `in` avalia se a primeira *string* está contida na segunda, retornando um booleano

③ A função `len()` pode ser utilizada com qualquer objeto, e ela retorna o seu tamanho, a implementação específica depende de cada classe, no caso das *strings*, será o número de caracteres

④ O método `lower()` retorna uma nova *string* com todos os caracteres em minúsculo

⑤ O método `upper()` retorna uma nova *string* com todos os caracteres em maiúsculo

⑥ O método `split()` retorna uma nova lista de *strings*, cada elemento contendo uma palavra da *string* original



Existem muitos métodos disponíveis, além da possibilidade de uso da função `help()`, temos a documentação original com todas as opções: [String Methods](#).

4.7. Listas

Um dos tipos mais versáteis em Python são as listas (`list`), comparado com outras linguagens uma lista é similar a uma `array`, porém ela vai muito além disso. As listas não são tipadas, ou seja cada elemento pode ser de um tipo diferente, além disso existe o conceito de *slicing* que permite formas extremamente poderosas de acesso aos seus elementos. Vamos a alguns exemplos.

Exercício 10 - Listas

listas.py

```
>>> lista = []
>>> type(lista)
<class 'list'>
>>> dir(lista) ①
[..., 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> len(lista) ②
0
>>> lista.append(1) ③
>>> lista.append(5)
>>> lista
[1, 5]
>>> len(lista)
2
>>> nova_lista = lista + ['Juracy', 'Leonardo', 3.1415] ④
>>> nova_lista
[1, 5, 'Juracy', 'Leonardo', 3.1415]
>>> nova_lista.insert(0, 'Zero') ⑤
>>> nova_lista
['Zero', 1, 5, 'Juracy', 'Leonardo', 3.1415]
>>> nova_lista.remove(5) ⑥
>>> nova_lista
['Zero', 1, 'Juracy', 'Leonardo', 3.1415]
```

- ① Lista dos membros (*métodos, constantes, ...*) disponíveis para o tipo `list`, uma dica é utilizar a função `help(list)` que detalhará melhor cada um deles
- ② A função `len()` pode ser utilizada com qualquer objeto, e ela retorna o seu tamanho, a implementação específica depende de cada classe, no caso das listas, será o número de elementos
- ③ O método `append()` inclui um novo elemento na lista
- ④ O uso do operador `+` com duas listas irá retornar uma nova lista juntando o conteúdo da primeira com a segunda (*sem alterar nenhuma delas*)
- ⑤ O método `insert()` inclui um novo elemento em uma posição específica da lista, lembrando que a primeira posição começa em 0
- ⑥ O método `remove()` retira um elemento da lista baseado no seu conteúdo, e não no seu índice

Exercício 11 - Indexação das Listas

listas_index.py

```
>>> lista = [1, 5, 'Juracy', 'Leonardo', 3.1415]
>>> lista.index('Juracy') ①
2
>>> lista.index(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 42 is not in list ②
>>> lista[2]
'Juracy'
>>> 1 in lista ③
True
>>> 'Juracy' in lista
True
>>> 'João' in lista
False
>>> lista[0]
'Zero'
>>> lista[4]
3.1415
>>> lista[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range ④
>>> lista[-1] ⑤
3.1415
>>> lista[-5]
1
```

- ① O método `index` retorna o índice de um elemento indicado
- ② Executar o método `index` com um valor não pertencente a lista retornará um `ValueError`
- ③ O operador `in` retorna se um objeto está contido na lista
- ④ Ao tentar acessar um índice inexistente na lista, normalmente maior que o número de elementos, o Python levanta uma exceção do tipo `IndexError`
- ⑤ É possível acessar elementos através de uma indexação negativa, sendo o último elemento `-1`, o penúltimo `-2` e assim por diante

Exercício 12 - Fatiamento de Listas

listas_slicing.py

```
>>> lista = [1, 5, 'Juracy', 'Leonardo', 3.1415]
>>> lista[1:3] ①
[5, 'Juracy']
>>> lista[1:-1] ②
[5, 'Juracy', 'Leonardo']
>>> lista[1:] ③
[5, 'Juracy', 'Leonardo', 3.1415]
>>> lista[:-1] ④
[1, 5, 'Juracy', 'Leonardo']
>>> lista[:] ⑤
[1, 5, 'Juracy', 'Leonardo', 3.1415]
>>> lista[::-2] ⑥
[1, 'Juracy', 3.1415]
>>> lista[::-1] ⑦
[3.1415, 'Leonardo', 'Juracy', 5, 1]
>>> del lista[2] ⑧
>>> lista
[1, 5, 'Leonardo', 3.1415]
>>> del lista[1:] ⑧
>>> lista
[1]
```

- ① O recurso de acesso aos elementos da lista aceita um segundo parâmetro separado por `:`, com isso ele retornará uma nova lista, começando a partir do elemento indicado até o elemento anterior ao segundo parâmetro
- ② Podemos também utilizar índices negativos, neste caso retorna uma nova lista a partir do segundo elemento até o penúltimo
- ③ Deixando o segundo parâmetro em branco significa até o final (*incluindo o último*)
- ④ O primeiro parâmetro em branco equivale a 0, ou seja, primeiro elemento
- ⑤ Retorna uma nova lista com todos os elementos da original, útil para executar cópias de uma lista
- ⑥ Um terceiro parâmetro pode ser informado como *step*, o *default* é 1, neste caso depois de pegar o primeiro elemento ele pulará dois (em vez de um) e assim seguirá até o fim da faixa
- ⑦ É possível também informar um *step* negativo, indicando que a nova lista começará a partir do último elemento da faixa até o primeiro, pulando de um em um (*poderia ser outro número também*)
- ⑧ A instrução `del` permite a remoção de elementos de uma lista através do seu índice ou fatiamento



O `del` permite diversos tipos de liberação, como destruição de objetos, remoção de chaves em um dicionário ou elementos numa lista.

4.8. Tuplas

As tuplas são similares a lista porém elas são imutáveis, e portanto não podem receber alterações. Existem algumas situações em que uma tupla pode ser preferida a uma lista, uma delas é como chave em um dicionário, como veremos na seção [Dicionários](#).

Exercício 13 - Tuplas

tuplas.py

```
>>> tupla = tuple() ①
>>> tupla = () ②
>>> type(tupla)
<class 'tuple'>
>>> dir(tupla) ③
[..., 'count', 'index']
>>> tupla = ('um') ④
>>> type(tupla)
<class 'str'>
>>> tupla = ('um',)
>>> type(tupla)
<class 'tuple'>
>>> tupla[0] ⑥
'um'
>>> cores = ('verde', 'amarelo', 'azul', 'branco')
>>> cores[0]
'verde'
>>> cores[-1]
'branco'
>>> cores[1:] ⑦
('amarelo', 'azul', 'branco')
```

① Tupla vazia, a partir da chamada da classe tuple

② Tupla vazia, representada por um ()

③ Diferente das listas, as tuplas possuem poucos métodos, e estes funcionam de forma similar às listas

④ ⚪ Erro muito comum, em Python esta expressão utilizando parenteses é tratada apenas como precedência, o que acaba atribuindo apenas a string 'um' a variável

⑤ Sintaxe correta para uma tupla de um elemento

⑥ Indexação similar as listas

⑦ Fatiamento similar as listas



O conceito de imutabilidade é bastante explorado em programação funcional, paradigma este que o Python possui algum suporte. Temos um bom exemplo disso em [Exercício 38 - Fibonacci recursivo](#).

4.9. Dicionários

Um outro tipo que faz parte dos alicerces do Python são os dicionários (`dict`), comparado com outras linguagens uma dicionário é similar a um `HashMap`, ou uma `array` associativa no PHP, e vai muito além disso.

Um dicionário é algo similar a uma lista de chave e valor, mas sem ordenação, por que as chaves são transformadas em *hashes* por questão de performance.

Assim como as listas, os dicionários não são tipados, nem a chave, nem o valor. Mas a chave precisa ser de um tipo imutável (como `str`, `int`, `float` ou `tuple`) ou seja cada elemento pode ser de um tipo diferente. Vamos a alguns exemplos.

Exercício 14 - Dicionários

dicionarios.py

```
>>> pessoa = {'nome': 'Juracy Filho', 'idade': 43, 'cursos': ['docker', 'python']}
>>> type(pessoa)
<class 'dict'>
>>> dir(dict) ①
[..., 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
>>> len(pessoa) ②
3
>>> pessoa
{'nome': 'Juracy Filho', 'idade': 43, 'cursos': ['docker', 'python']}
>>> pessoa['nome'] ③
'Juracy Filho'
>>> pessoa['idade']
43
>>> pessoa['cursos']
['docker', 'python']
>>> pessoa['tags']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'tags' ④
>>> pessoa.keys() ⑤
dict_keys(['nome', 'idade', 'cursos'])
>>> pessoa.values() ⑥
dict_values(['Juracy Filho', 43, ['docker', 'python']])
>>> pessoa.items() ⑦
dict_items([('nome', 'Juracy Filho'), ('idade', 43), ('cursos', ['docker', 'python'])])
>>> pessoa.get('idade') ⑧
43
>>> pessoa.get('tags')
>>> pessoa.get('tags', [])
[]
```

- ① Lista dos membros (*métodos, constantes, ...*) disponíveis para o tipo `dict`, uma dica é utilizar a função `help(dict)` que detalhará melhor cada um deles
- ② A função `len()` pode ser utilizada com qualquer objeto, e ela retornar o seu tamanho, a implementação específica depende de cada classe, no caso dos dicionários, será o número de elementos (chave e valor)
- ③ O uso do recurso de indexação também funciona nos dicionários usando a chave do valor que se quer encontrar
- ④ Ao tentar recuperar um valor de uma chave inexistente através do índice o Python gera uma exceção do tipo `KeyError`
- ⑤ O método `keys()` retorna uma espécie de lista (`dict_keys`) com todas as chaves
- ⑥ O método `values()` retorna uma espécie de lista (`dict_values`) com todos os valores
- ⑦ O método `items()` retorna uma espécie de lista (`dict_items`) com todos as chaves e valores, em algo similar a uma lista de tuplas
- ⑧ O método `get()` funciona de forma similar ao índice, porém caso a chave não exista retorna um `None`. Também é possível colocar um segundo parâmetro com o valor a ser retornado caso a chave não exista

Exercício 15 - Atualização nos Dicionários

dicionarios_update.py

```
>>> pessoa = {'nome': 'Juracy Filho', 'idade': 43, 'cursos': ['docker', 'python']}
>>> pessoa['idade'] = 44 ①
>>> pessoa['cursos'].append('angular')
>>> pessoa
{'nome': 'Juracy Filho', 'idade': 44, 'cursos': ['docker', 'python', 'angular']}
>>> pessoa.pop('idade') ②
44
>>> pessoa
{'nome': 'Juracy Filho', 'cursos': ['docker', 'python', 'angular']}
>>> pessoa.update({'idade': 40, 'Sexo': 'M'}) ③
>>> pessoa
{'nome': 'Juracy Filho', 'idade': 40, 'cursos': ['docker', 'python'], 'Sexo': 'M'}
>>> del pessoa['cursos'] ④
>>> pessoa
{'nome': 'Juracy Filho', 'idade': 40, 'Sexo': 'M'}
>>> pessoa.clear() ⑤
>>> pessoa
{}
```

- ① É possível alterar o valor de uma determinada chave, se a chave não existir ela será criada
- ② O método `pop()` retorna o valor de uma determinada chave e a remove do dicionário
- ③ O método `update()` recebe um outro dicionário e atualiza o objeto principal (*merge*)
- ④ O `del` também permite remover elementos através da sua chave
- ⑤ O método `clear()` limpa completamente o dicionário

Habilidades adquiridas

Conhecendo os recursos básicos da linguagem temos uma plataforma sólida para mergulhar nos mais diversos recursos, sempre buscando uma abordagem mais pythônica.

- Instruções:

`del`

Liberação ou remoção de objetos.

TODO

5. Ninho de cobras

Apesar do interpretador python ser fantástico, em aplicações reais precisamos escrever nosso código em módulos, para isso precisamos de editor de texto, não necessariamente um IDE, neste curso utilizamos o Microsoft Visual Source Code (**vscode** para os íntimos).

Um módulo python é um arquivo contendo instruções Python. Normalmente ele deve ter a extensão .py, e se for utilizado por outro módulo precisa ter um nome válido como um identificador python, nada de traços por exemplo.

Referência: [Python Docs — Modules](#)

5.1. Primeiro módulo

Exercício 16 - Área do círculo - versão 1

area_circulo_v1.py

```
pi = 3.1415926
raio = 15
print('Área do círculo', pi * raio ** 2)
```

Para executar o exemplo acima: `python area_circulo_v1.py`

5.2. Encoding

Executar o exemplo anterior com o Python 2 pode gerar um erro de *encoding*: `SyntaxError: Non-ASCII character '\xc3' in file area_circulo_v1.py on line 3, but no encoding declared; see http://python.org/dev/peps/pep-0263/ for details`

Este comportamento é totalmente esperado já que o *encoding* padrão do Python 2 é ASCII e os acentos encontrados nas palavras área e círculo não são cobertos pelo ASCII, já no Python 3 o padrão é UTF-8 e os caracteres que mais utilizamos são totalmente cobertos.

Conforme a mensagem de erro indica podemos consultar a [PEP 0263](#) para uma explicação completa.

Vejamos a solução aplicando o *encoding*.

Exercício 17 - Área do círculo - versão 2

area_circulo_v2.py

```
# -*- coding: utf-8 -*-
pi = 3.1415926
raio = 15
print('Área do círculo', pi * raio ** 2)
```



Apesar de não dar mais erro no Python 2, a própria sintaxe do `print` mudou um pouco, mas não vamos nos ater ao Python 2, a maior preocupação é a necessidade de escrever em outros encodings que não o UTF-8 por exemplo.

5.3. Shebang

É possível executar um módulo python sem chama-lo explicitamente na linha de comando através do **shebang** que é um comentário na primeira linha do módulo.

Referência: [Wikipedia — Shebang](#)

Exercício 18 - Área do círculo - versão 3

`area_circulo_v3.py`

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

pi = 3.1415926
raio = 15
print('Área do círculo', pi * raio ** 2)
```

Antes de poder executá-lo em ambientes **unix** em geral é necessário dar direito a execução: `chmod a+x area_circulo_v3.py`

Após isso podemos executá-lo diretamente: `./area_circulo_v3.py`



É muito comum na programação de *scripts* deixar o módulo principal sem a extensão, simplificado sua execução, por exemplo, considerando que o *script* está no **path** a chamada seria simplesmente: `area_circulo_v3`

5.4. Baterias inclusas

O Python também é conhecido por vir com baterias inclusas, isso é atribuído a sua extensão biblioteca padrão e a algumas bibliotecas externas incorporadas ao longo do tempo. Para a maior parte das necessidades corriqueiras, a próprio biblioteca padrão já atenderá.



Infelizmente é completamente inviável cobrir a biblioteca padrão. Até por que muitos destes recursos exigiriam um curso só para si. A maior parte dela é escrita em python mesmo, simplificando seu estudo. Porém por motivos de performance existem algumas implementadas em **C**.

Mas felizmente a documentação é vasta, tanto através do `__doc__` quanto da documentação oficial em <https://docs.python.org/3/library/index.html>.

Exercício 19 - Área do círculo - versão 4

area_circulo_v4.py

```
#!/usr/bin/python3
import math ①

raio = 15
print('π =', math.pi) ②
print('Área do círculo', math.pi * raio ** 2)
```

① Importação do módulo `math` da biblioteca padrão

② Uso da constante `pi` no módulo `math`



Em python, tudo é um objeto, inclusive funções, classes e módulos. Ao importar o módulo `math`, um novo identificador `math` da classe `module` fica disponível no escopo, seus membros são em sua maioria os identificadores disponíveis no módulo em questão.

5.5. Entrada de dados

Exercício 20 - Área do círculo - versão 5

area_circulo_v5.py

```
#!/usr/bin/python3
import math

raio = input('Informe o raio:') ①
print('π =', math.pi)
print('Área do círculo', math.pi * raio ** 2)
```

① A função `input` do `_builtins_` solicita ao usuário um entrada de dados (normalmente via teclado) e retorna uma **string**.

A execução do [Exercício 20 - Área do círculo - versão 5](#) irá resultar no erro abaixo:



```
Traceback (most recent call last):
  File "exercicios/modulos/area_circulo_v5.py", line 6, in <module>
    print('Área do círculo', math.pi * raio ** 2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Conforme já discutido o python não faz todo tipo de coerção de tipos automaticamente, apenas em casos específicos em que não exista redundância de possibilidades.

Neste caso `raio` é do tipo **string** e não pode ser usado na expressão matemática para o cálculo.

Exercício 21 - Área do círculo - versão 5 (correção)

area_circulo_v5_fix.py

```
#!/usr/bin/python3
import math

raio = int(input('Informe o raio:')) ①
print('π =', math.pi)
print('Área do círculo', math.pi * raio ** 2)
```

① Chamando a classe `int`, o resultado do `input` é convertido e o cálculo pode ser efetuado.

5.6. Nome do módulo

Exercício 22 - Área do círculo - versão 6

area_circulo_v6.py

```
#!/usr/bin/python3
import math

print('Nome do módulo', __name__) ①

raio = int(input('Informe o raio:'))
print('π =', math.pi)
print('Área do círculo', math.pi * raio ** 2)
```

① `__name__` é um dos atributos da classe `module`. E retorna o nome do módulo, a exceção desta regra ocorre quando o módulo não existe ou é o principal.

Executar o [Exercício 22 - Área do círculo - versão 6](#) das seguintes formas:

1. Chamando pela linha de comando
2. Tentando importar a partir do interpretador: `import area_circulo_v6`



Ao importar ele a partir do interpretador (ou até de outro módulo), todo o código do módulo é executado.

5.7. E se ...

Exercício 23 - Área do círculo - versão 7

area_circulo_v7.py

```
#!/usr/bin/python3
import math

if __name__ == '__main__': ①
    raio = int(input('Informe o raio:'))
    print('Área do círculo', math.pi * raio ** 2)
```

- ① Instrução `if` para execução condicional, sua sintaxe é `if <expressão>:`. O bloco de execução em python é definido através de indentação.



O bloco em python é definido pela indentação, que pode ser composta de espaços ou tabs, porém precisam ser consistentes no mesmo módulo, usando a mesma quantidade de espaços ou tabs. Nos nossos exemplos usaremos 4 espaços.

Referência: [PEP-8 — Tabs or Spaces?](#)

5.8. Quebrando nosso código em funções

No [Exercício 23 - Área do círculo - versão 7](#) ajustamos o nosso módulo para ser executado apenas através da chamada direta, porém ficamos sem nenhuma funcionalidade ao importa-lo. Agora podemos dividir nosso código em funções, permitindo que as funções possa ser executadas por outros módulos.

Exercício 24 - Área do círculo - versão 8

area_circulo_v8.py

```
#!/usr/bin/python3
import math

def circulo(raio): ①
    print('Área do círculo', math.pi * raio ** 2)

if __name__ == '__main__':
    raio = int(input('Informe o raio:'))
    circulo(raio) ②
```

- ① Instrução `def` para criação de uma função, sua sintaxe é `def <nome>(<parâmetros ...>):`. O bloco de execução é definido através de indentação.
- ② Chamada da nova função `circulo`, que recebe como parâmetro o raio a ser utilizado no cálculo.

Agora além de poder executá-lo via linha de comando, poderia ser utilizado no interpretador ou outro módulo da seguinte forma:



```
import area_circulo_v8  
area_circulo_v8.circulo(15)
```

Tentaremos sempre seguir a **PEP-8** neste material, note no exemplo acima que foram utilizados duas linhas em branco antes e depois da função `circulo`. Isso está definido em [PEP 8 — Blank Lines](#).



Existem diversas ferramentas para verificar e até ajustar o seu código para atender esta padronização.

Caso o módulo seja alterado depois que importado no interpretador podemos usar a função `reload` do módulo `imp` da biblioteca padrão.



```
import imp  
imp.reload(area_circulo_v8)
```

5.9. Funções podem retornar valores

Em Python, toda função tem um valor de retorno, se isso não for definido, o valor de retorno é `None`.

No [Exercício 24 - Área do círculo - versão 8](#) já podíamos chamar a função `circulo` e a mesma imprimia o resultado na tela, mas e se quiséssemos utilizar este valor para um novo cálculo, ou em vez de imprimir na tela, gravar em um arquivo?

Podemos usar a instrução `return` para indicar explicitamente um valor de retorno.

Exercício 25 - Área do círculo - versão 9

`area_circulo_v9.py`

```
#!/usr/bin/python3  
import math  
  
def circulo(raio):  
    return math.pi * raio ** 2  ①  
  
if __name__ == '__main__':  
    raio = int(input('Informe o raio:'))  
    area = circulo(raio) ②  
    print('Área do círculo', area) ③
```

① Instrução `return`, retornando apenas o resultado do cálculo.

② Armazena o valor de retorno na variável `area`

③ Imprime o resultado do cálculo (apenas quando chamado pela linha de comando)

Outro exemplo de uso:



```
import area_circulo_v9
calc = area_circulo_v9.circulo(15)
print('Resultado:', calc)
```

5.10. Passando argumentos pela linha de comando

Exercício 26 - Área do círculo - versão 10

area_circulo_v10.py

```
#!/usr/bin/python3
import math
import sys ①

def circulo(raio):
    return math.pi * raio ** 2

if __name__ == '__main__':
    raio = int(sys.argv[1]) ②
    area = circulo(raio)
    print('Área do círculo', area)
```

- ① Importação do módulo sys, um importante módulo da biblioteca padrão, algumas funções: argv, exit, getdefaultencoding, path, stderr, stdin, stdout.
- ② Substituição do input pelo primeiro argumento da linha de comando. O argv é uma lista de str que contém os argumentos da linha de comando, sendo que o primeiro elemento da lista o próprio script chamado, assim sendo o segundo elemento ([1]) é o primeiro argumento.



Exemplo de chamada do script: ./area_circulo_v10.py 15



Como não há nenhuma verificação no momento, chamar o script sem passar o argumento gera o seguinte erro: IndexError: list index out of range.

5.11. Verificação dos argumentos

Uma chamada incorreta do [Exercício 27 - Área do círculo - versão 11](#) deixaria o usuário bastante confuso, no próprio exercício veremos um tratamento melhor para isso.

Exercício 27 - Área do círculo - versão 11

area_circulo_v11.py

```
#!/usr/bin/python3
import math
import sys

def circulo(raio):
    return math.pi * raio ** 2

if __name__ == '__main__':
    if len(sys.argv) < 2: ①
        ②
        print("""\nÉ necessário informar o raio do círculo.

Sintaxe: area_circulo <raio>""")
    else: ③
        raio = int(sys.argv[1])
        area = circulo(raio)
        print('Área do círculo', area)
```

- ① Verificação do tamanho da lista de argumentos, caso não tenha pelo menos 2, imprimir uma ajuda.
- ② Uso dos recursos de string de múltiplas linhas e contra-barra para não gerar uma linha extra.
- ③ Instrução else do if, permitindo um fluxo alternativo caso a condição não tenha sido atingida, neste caso ter 2 ou mais elementos na lista.

5.12. Melhorando um pouco o nosso *help*

Exercício 28 - Área do círculo - versão 12

area_circulo_v12.py

```
#!/usr/bin/python3
import math
import sys

def circulo(raio):
    return math.pi * raio ** 2

def help(): ①
    print("""
É necessário informar o raio do círculo.

Sintaxe: area_circulo <raio>""")

if __name__ == '__main__':
    if len(sys.argv) < 2:
        help() ②
    else:
        raio = int(sys.argv[1])
        area = circulo(raio)
        print('Área do círculo', area)
```

① Nova função *help* para impressão da ajuda.

② Chamada do *help*, tornando o código mais legível.

5.13. Dando retorno ao sistema operacional

Scripts podem indicar um nível de erro ao sistema operacional, em Python quando não definido o retorno padrão é 0, que significa que foi executado com sucesso.

No módulo *sys* tem uma função chamada *exit* que permite indicar o nível de erro e encerrar a execução imediatamente.

Exercício 29 - Área do círculo - versão 13

area_circulo_v13.py

```
#!/usr/bin/python3
import math
import sys

def circulo(raio):
    return math.pi * raio ** 2

def help():
    print("""\
É necessário informar o raio do círculo.

Sintaxe: area_circulo <raio>""")

if __name__ == '__main__':
    if len(sys.argv) < 2: ①
        help()
        sys.exit(1)

    raio = int(sys.argv[1])
    area = circulo(raio)
    print('Área do círculo', area)
```

- ① Caso não tenha argumentos suficientes encerra a execução e retorna ao S0 nível de erro 1.
Com isso não é mais necessário o else.

5.14. Validando argumentos

Podemos garantir também que o argumento passado seja um número inteiro, dando mais robustez ao nosso projeto.

Exercício 30 - Área do círculo - versão 14

area_circulo_v14.py

```
#!/usr/bin/python3
import math
import sys

def circulo(raio):
    return math.pi * raio ** 2

def help():
    print("""
É necessário informar o raio do círculo.

Sintaxe: area_circulo <raio>""")

if __name__ == '__main__':
    if len(sys.argv) < 2:
        help()
        sys.exit(1)

    if not sys.argv[1].isnumeric(): ①
        help()
        print('O raio deve ser um valor inteiro')
        sys.exit(2)

    raio = int(sys.argv[1])
    area = circulo(raio)
    print('Área do círculo', area)
```

- ① Se o primeiro argumento não for numérico, imprimir a ajuda, explicar o erro e sair com código 2.

Habilidades adquiridas

Neste capítulo realizamos um exercício simples de forma progressiva para o cálculo da área de um círculo e adquirimos os seguintes conceitos e habilidades:

- Criação de módulos e seu execução;
- Possíveis questões referentes ao *encoding* do código fonte;
- Execução de módulos como *scripts*;
- Importação de funções e uso da biblioteca padrão (módulos *sys* e *math*);
- Conceitos simples de entrada de dados;
- O conceito de módulo como objeto;
- Execução condicional simples, a **importância da indentação** e o que é o **PEP 8**;
- Criação de funções, com ou sem retorno de dados;
- Argumentos através da linha de comando. **Essencial para scripts**;
- Verificações da entrada de dados (através de condicionais e expressões), como número de argumentos e tipo dos mesmos;

- Como indicar sucesso ou falha no *script* para o sistema operacional.

Desafio

Incluir no exercício a opção do cálculo de área de um quadrado (todos lados iguais), que utiliza a seguinte fórmula: L^2

Sem perder a função do área do círculo.



Incluir um primeiro parâmetro obrigatório: **circulo** ou **quadrado**.

Exemplo de solução disponível em [Desafio 1 - Cálculo da área do círculo ou quadrado](#).

6. Instruções

6.1. Conhecendo o while

Exercício 31 - Fibonacci - While infinito

fibonacci_v1.py

```
#!/usr/bin/python3

def fibonacci():
    penultimo = 0
    ultimo = 1

    print(penultimo)
    print(ultimo)

    while True: ①
        proximo = penultimo + ultimo
        print(proximo)
        penultimo = ultimo
        ultimo = proximo

if __name__ == '__main__':
    fibonacci()
```

① Instrução while com uma condição sempre verdadeira.

6.2. Execute enquanto ...

Exercício 32 - Fibonacci - While condicional

fibonacci_v2.py

```
#!/usr/bin/python3

def fibonacci(limite):
    penultimo = 0
    ultimo = 1

    print(penultimo)
    print(ultimo)

    while ultimo < limite: ①
        proximo = penultimo + ultimo
        print(proximo)
        penultimo = ultimo
        ultimo = proximo

if __name__ == '__main__':
    fibonacci(1000)
```

① Instrução while com um limitador.

6.3. Packing

Exercício 33 - Fibonacci - Uso do packing

fibonacci_v3.py

```
#!/usr/bin/python3

def fibonacci(limite):
    penultimo = 0
    ultimo = 1

    print(penultimo)
    print(ultimo)

    while ultimo < limite:
        penultimo, ultimo = ultimo, penultimo + ultimo ①
        print(ultimo)

if __name__ == '__main__':
    fibonacci(1000)
```

- ① Uso do recurso de *packing*, atribuindo duas variáveis ao mesmo tempo

6.4. Conhecendo o for

Exercício 34 - Fibonacci - Iterando uma lista

fibonacci_v4.py

```
#!/usr/bin/python3

def fibonacci(limite):
    resultado = [0, 1] ①

    while resultado[-1] < limite: ②
        resultado.append(resultado[-2] + resultado[-1]) ③

    return resultado ④

if __name__ == '__main__':
    for fib in fibonacci(1000): ⑤
        print(fib)
```

- ① Criando uma lista com os valores iniciais
② Testando o último elemento da lista
③ Somando os dois últimos elementos e adicionando na lista
④ Retornando a lista
⑤ Iterando a lista através da instrução *for*.

6.5. Totalizando uma lista

Exercício 35 - Fibonacci - sum

fibonacci_v5.py

```
#!/usr/bin/python3

def fibonacci(limite):
    resultado = [0, 1]

    while resultado[-1] < limite:
        resultado.append(sum(resultado[-2:])) ①

    return resultado

if __name__ == '__main__':
    for fib in fibonacci(1000):
        print(fib)
```

① Uso da função `sum` do `__builtins__`, totalizando os dois últimos elementos

6.6. Forçando a quebra de um laço

Exercício 36 - Fibonacci - break

fibonacci_v6.py

```
#!/usr/bin/python3

def fibonacci(quantidade):
    resultado = [0, 1]

    while True: ①
        resultado.append(sum(resultado[-2:]))

    if len(resultado) == quantidade: ②
        break ③

    return resultado

if __name__ == '__main__':
    # Listar os 20 primeiros números da sequência ④
    for fib in fibonacci(20):
        print(fib)
```

- ① Uso de um laço infinito (para quebra forçada)
- ② Uso do if para avaliar a nova condição de número de elementos
- ③ Instrução break, que pode ser usada em while e for.
- ④ Comentário

É possível construir o mesmo exemplo sem o uso do break, usando a nova condição adaptada no while:



```
while len(resultado) < limite:
```

6.7. Gerando uma lista de números

Exercício 37 - Fibonacci - range

fibonacci_v7.py

```
#!/usr/bin/python3

def fibonacci(quantidade):
    resultado = [0, 1]

    for i in range(2, quantidade): ①
        resultado.append(sum(resultado[-2:]))

    return resultado

if __name__ == '__main__':
    # Listar os 20 primeiros números da sequência
    for fib in fibonacci(20):
        print(fib)
```

- ① O `range` gera uma lista de inteiro (na realidade um iterator), neste caso dos número 2 a 19, totalizando 18 elementos, que adicionados à lista inicial perfazem 20 números da sequência de *fibonacci*.

6.8. Recursão

Exercício 38 - Fibonacci recursivo

fibonacci_recursive_v1.py

```
#!/usr/bin/python3

def fibonacci(quantidade, sequencia=(0, 1)): ①
    # Importante: Condição de parada
    if len(sequencia) == quantidade: ②
        return sequencia
    return fibonacci(quantidade, sequencia + (sum(sequencia[-2:]),)) ③ ④ ⑤

if __name__ == '__main__':
    # Listar os 20 primeiros números da sequência
    for fib in fibonacci(20):
        print(fib)
```

- ① Uso do recurso de valor *default* nos parâmetros
- ② Condição de parada. Toda a função recursiva deve ter uma condição de parada
- ③ Uso preferido de imutabilidade, não fazendo nenhuma alteração e sim gerando um novo objeto *tuple*
- ④ Uso de concatenação de tuplas, para manter a imutabilidade, no lugar de listas com *append*
- ⑤ Tuplas com um elemento devem ter uma vírgula extra, senão é interpretado como precedência de operadores

Pegadinha (*pitfall*)



O uso de valores defaults em funções usando tipo mutáveis é desaconselhado, haja visto que este objeto é criado uma única vez (junto com a criação da função) e este valor é reaproveitado toda vez que o parâmetro não é passado.

Isso significa que uma mudança neste parâmetro pode afetar chamadas posteriores da mesma função quando o parâmetro não for informado!



O uso de recursividade deve ser feito com parcimônia, muita atenção e sempre ter uma condição de parada.

6.9. Operador ternário ... if ... else ...

Exercício 39 - Fibonacci recursivo com operador ternário

fibonacci_recursive_v2.py

```
#!/usr/bin/python3

def fibonacci(quantidade, sequencia=(0, 1)):
    # Importante: Condição de parada ① ②
    return sequencia if len(sequencia) == quantidade else \
        fibonacci(quantidade, sequencia + (sum(sequencia[-2:]),))

if __name__ == '__main__':
    # Listar os 20 primeiros números da sequência
    for fib in fibonacci(20):
        print(fib)
```

① Uso do operador ternário if ... else

② Uso de contra-barra para continuar linha

Habilidades adquiridas 🎓

Através da sequência de fibonacci, conseguimos evoluir na nossa compreensão dos recursos da linguagem, entre eles:

- Instruções

while

Laço condicional;

for

Iteração (laço) em uma coleção/iterável;

break

Força a saída imediata do laço mais interno (for or while);

- Funções do __builtins__

sum

Itera um objeto somando seus elementos (podendo receber uma função para ajustar a lógica de acúmulo);

range

Gerar uma sequência de números (na realidade um generator);

- Um pouco do conceito de *packing* e *unpacking*.
- Uso de recursão, tuplas e imutabilidade;
- Uso do operador ternário;
- Continuação de linha;

Desafio

Criar *script* para informar se um determinado número faz parte ou não da sequência de *fibonacci*, recebendo este número da linha de comando.



A sequência deve parar de gerar números assim que passar ou igualar o número informado.

Exemplo de solução disponível em [Desafio 2 - Fibonacci](#)

Quero mais

Existem mais duas versões avançadas de *fibonacci* nos anexos:

- [Exemplo Avançado 1 - Fibonacci recursivo decrescente sem memoize](#)
- [Exemplo Avançado 2 - Fibonacci recursivo com decorators, memoize e classes](#)

7. Manipulação de arquivos

Entra filhote, sai cobra criada.

Exercício 40 - Leitura Básica de Arquivo

io_v1.py

```
#!/usr/bin/python3

arquivo = open('pessoas.csv')
dados = arquivo.read()
arquivo.close()

for registro in dados.splitlines():
    print('Nome: {} Idade: {}'.format(*registro.split(',')))
```

Exercício 41 - Leitura Básica de Arquivo (*stream*)

io_v2.py

```
#!/usr/bin/python3

arquivo = open('pessoas.csv')
for registro in arquivo:
    print('Nome: {} Idade: {}'.format(*registro.split(',')))
arquivo.close()
```

Exercício 42 - Leitura Básica de Arquivo (*stream*) — Fix

io_v3.py

```
#!/usr/bin/python3

arquivo = open('pessoas.csv')
for registro in arquivo:
    print('Nome: {} Idade: {}'.format(*registro.strip().split(',')))
arquivo.close()
```

Exercício 43 - Mais robustez com try..finally

io_v4.py

```
#!/usr/bin/python3

try: ①
    arquivo = open('pessoas.csv')
    for registro in arquivo:
        print('Nome: {} Idade: {}'.format(*registro.strip().split(',')))
finally: ②
    arquivo.close() ②

if arquivo.closed:
    print('Arquivo já foi fechado!')
```

① TODO: try..finally

② TODO: sempre fechar o arquivo para liberar quaisquer recursos associados. GC..

Exercício 44 - Leitura de Arquivo com with

io_v5.py

```
#!/usr/bin/python3

with open('pessoas.csv') as arquivo: ①
    for registro in arquivo:
        print('Nome: {} Idade: {}'.format(*registro.strip().split(',')))

if arquivo.closed: ②
    print('Arquivo já foi fechado!')
```

① TODO: with..as, __enter__, __exit__

② TODO: propriedade closed que indica se o arquivo está fechado

Exercício 45 - Gravação de Arquivo

io_v6.py

```
#!/usr/bin/python3

with open('pessoas.csv') as entrada:
    with open('pessoas.txt', 'w') as saida: ①
        for registro in entrada:
            pessoa = registro.strip().split(',')
            print('Nome: {} Idade: {}'.format(*pessoa), file=saida)
```

① TODO: O open suporta um segundo parâmetro para indicar o modo de abertura do arquivo, o default é 'r' que é para leitura, aqui usamos o 'w' para abri-lo para gravação

Exercício 46 - Leitura de Arquivo com o módulo CSV

io_csv.py

```
import csv ①

with open('pessoas.csv') as entrada:
    for pessoa in csv.reader(entrada): ②
        print('Nome: {} Idade: {}'.format(*pessoa))
```

① TODO: Módulo csv da biblioteca padrão para tratamento de arquivos CSV

② TODO: reader → iterator

Habilidades adquiridas 🎓

Manipulação básica de arquivos texto, incluindo ainda algumas habilidades extras:

- Instruções:

try...finally

Bloco de finalização para fins como liberação de recursos, inclusive em caso de exceções;

with...as

Criação de um bloco associado ao contexto de um objeto, com inicialização `__enter__` e finalização `__exit__` providas pela classe, já com o suporte embutido ao `try...finally`. O `as` é opcional, e é usado para atribuir o objeto a uma nova variável;

- Métodos especiais (ou mágicos):

`__enter__`

Entrada do contexto do objeto indicado pelo `with`;

`__exit__`

Saída do contexto do bloco `with`;

- Uso da classe `File`, na realidade um conjuntos de classes gerenciadas automaticamente pelo módulo `io`;
- Uso do `str.format`;
- Mais uso de *unpacking*;
- Uso do módulo `csv`.

Desafio 🏆

Extraír o nono e o quarto campos do arquivo CSV sobre **Região de influência das Cidades** do IBGE, que pode ser baixado em: http://www.geoservicos.ibge.gov.br/geoserver/wms?service=WFS&version=1.0.0&request=GetFeature&typeName=CGEO:RedeUrbanaSintese_Regic2007&outputFormat=CSV. ignorando a primeira linha que é o cabeçalho:



O arquivo se encontra em ISO-8859-1 (*aka latin1*), será necessário usar o parâmetro encoding da função open.

Exemplo de solução disponível em [Desafio 3 - Tratamento de CSV](#), temos também um exemplo mais avançado baixando o arquivo diretamente da internet em [Exemplo Avançado 3 - Tratamento de CSV com download](#).

8. Comprehension

Exercício 47 - Dobros

comprehension_v1.py

```
#!/usr/bin/python3

dobros = [i * 2 for i in range(10)]
print(dobros)
```

Exercício 48 - Dobros dos pares

comprehension_v2.py

```
#!/usr/bin/python3

dobros_dos_pares = [i * 2 for i in range(10) if i % 2 == 0]
print(dobros_dos_pares)
```

Exercício 49 - Generators

comprehension_v3.py

```
#!/usr/bin/python3

generator = (i * 2 for i in range(10) if i % 2 == 0)
print(next(generator)) # Saída 0
print(next(generator)) # Saída 4
print(next(generator)) # Saída 8
print(next(generator)) # Saída 12
print(next(generator)) # Saída 16
print(next(generator)) # Gera uma exceção, indicando que acabou
```

Exercício 50 - Generators com for

comprehension_v4.py

```
#!/usr/bin/python3

generator = (i * 2 for i in range(10) if i % 2 == 0)

for numero in generator:
    print(numero)
```

Exercício 51 - Dict Comprehension

comprehension_v5.py

```
#!/usr/bin/python3

dicionario = {i: i * 2 for i in range(10) if i % 2 == 0}

print(dicionario)

for numero, dobro in dicionario.items():
    print(f'{numero} x 2 = {dobro}')
```

Habilidades adquiridas 🎓

Uso de instruções `for` e `if` em listas e dicionários (*list comprehension*), incluindo ainda algumas habilidades extras:

- Conceito e uso de *generators*;
- Uso do f-string.

Desafio 💪

Listar toda a tabuada de multiplicação de 1 a 9 usando *list comprehension*, em apenas uma linha. Não é válido utilizar linha de código separadas com : (dois pontos).



É possível utilizar `for` aninhado no *list comprehension*.

Exemplo de solução disponível em [Desafio 4 - Tabuada](#).

9. Programação funcional

Python é uma linguagem multi-paradigma, cobrindo programação estruturada, imperativa, orientação a objetos e programação funcional (tendo **Lisp** e **Haskell** como influências).

A programação funcional em Python sempre foi muito discutida, até por que o seu criador sempre achou alguns desses conceitos complexos e a linguagem tinha o intuito de ser simples. E com o advento dos *lists comprehensions* na versão 2.0, a parte funcional ficou menos relevante, já que passou a ter uma sintaxe mais simples para alguns desses conceitos.

Ainda assim, o arsenal existente é bastante interessante como veremos a seguir e muitos desses recursos se integram transparentemente com a linguagem, tornando-a tão poderosa.

9.1. Capacidades implementadas

Temos vários recursos conhecidamente de linguagens funcionais, entre eles:

- *First Class Functions*
- *High Order Functions*
- *Anonymous Functions*
- *Closure*
- *Recursion*
- *Immutability*
- *Lazy Evaluation*



Explicaremos em detalhes a seguir, mas mantivemos aqui os termos em inglês por acha-los mais adequados, as traduções ora ficam meio sem sentido.

9.2. *First Class Functions* - Funções de primeira classe

Capacidade de usar as funções como entidades de primeira classe, em variáveis por exemplo.

Exercício 52 - Funções de primeira classe

first_class_functions.py

```
#!/usr/bin/python3

def dobro(x):
    return x * 2

def quadrado(x):
    return x ** 2

if __name__ == '__main__':
    # Retornar alternadamente o dobro ou quadrado nos números de 1 a 10
    funcs = [dobro, quadrado] * 5 ①
    for func, numero in zip(funcs, range(1, 11)): ②
        print(f'0 {func.__name__} de {numero} é {func(numero)}') ③ ④
```

① TODO: Operador de multiplicação em listas

② TODO: zip

③ TODO: function.__name__

④ TODO: function.__call__

9.3. High Order Functions - Funções de alta ordem

Capacidade de uma função de receber como parâmetro e/ou retornar outras funções.

Exercício 53 - Funções de alta ordem

high_order_functions.py

```
#!/usr/bin/python3
from first_class_functions import dobro, quadrado ①

def process(titulo, lista, funcao): ②
    print(f'Processando: {titulo}')
    for i in lista:
        print(i, '=>', funcao(i)) ③

if __name__ == '__main__':
    process('Dobros de 1 a 10', range(1, 11), dobro) ④
    process('Quadrados de 1 a 10', range(1, 11), quadrado) ④
```

① TODO: Aproveitando funções dobro e quadrado

② TODO: recebendo função

③ TODO: chamando a função recebida

④ TODO: passando uma função como parâmetro

9.4. Closure - Funções com escopos aninhados

Funções que podem ser aninhadas e ter acesso ao escopo da função na qual foi definida, inclusive impedindo o [Garbage Colector](#) de liberá-las.

Exercício 54 - Funções com escopos aninhados (closure)

`closure.py`

```
#!/usr/bin/python3

def multiplier(times):
    def calc(x): ①
        return x * times ②
    return calc ③

if __name__ == '__main__':
    dobro = multiplier(2)
    triplo = multiplier(3)

    print(dobro, triplo) ④

    print(f'0 triplo de 3 é {triplo(3)}') ⑤
    print(f'0 dobro de 7 é {dobra(7)}') ⑥
    print(f'0 dobro de 3 é {dobra(3)}') ⑥
```

① TODO: *Closure em si* (função aninhada com acesso a `times`)

② TODO: Execução real acessa o `times`

③ TODO: retorna da função aninhada

④ TODO: conhecendo um pouco das funções criadas

⑤ TODO:

⑥ TODO:

9.5. Anonymous Functions - Funções anônimas (lambda)

O *lambda* (no alfabeto grego λ) é baseado num conceito matemático e computacional chamado de *lambda calculus* e sua sintaxe é quase uma cópia da sintaxe do [Lisp](#), na qual foi baseada.

Na prática são funções anônimas, que nem precisam ter um identificador definido. Em Python podemos utilizá-las através do `lambda`.

It may seem perverse to use lambda to introduce a procedure/function. The notation goes back to **Alonzo Church**, who in the 1930's started with a "hat" symbol; he wrote the square function as " $\hat{y} . y \times y$ ". But frustrated typographers moved the hat to the left of the parameter and changed it to a capital lambda: " $\Lambda y . y \times y$ "; from there the capital lambda was changed to lowercase, and now we see " $\lambda y . y \times y$ " in math books and `(lambda (y) (* y y))` in Lisp. If it were up to me, I'd use fun or maybe ^

— Peter Norvig, <http://norvig.com/lispy2.html>

Exercício 55 - Totalização de compras (lambda)

`lambda_functions.py`

```
#!/usr/bin/python3

compras = ( ①
    {'quantidade': 2, 'preco': 10},
    {'quantidade': 3, 'preco': 20},
    {'quantidade': 5, 'preco': 14},
)
totais = tuple( ②
    map( ③
        lambda compra: compra['quantidade'] * compra['preco'], ④
        compras
    )
)
print('Preços totais:', list(totais))
print('Total geral:', sum(totais)) ⑤
```

① TODO:

② TODO:

③ TODO:

④ TODO:

⑤ TODO:

Exercício 56 - Totalização de compras sem o uso de lambda

lambda_functions_alternative.py

```
#!/usr/bin/python3

def calc_preco_total(compra): ①
    return compra['quantidade'] * compra['preco']

compras = (
    {'quantidade': 2, 'preco': 10},
    {'quantidade': 3, 'preco': 20},
    {'quantidade': 5, 'preco': 14},
)
totais = tuple(
    map(
        calc_preco_total, ②
        compras
    )
)
print('Preços totais:', list(totais))
print('Total geral:', sum(totais))
```

① TODO:

② TODO:

9.6. Recursion - Recursividade

Funções recursivas (que chamam a si mesmas) são bastante comuns nas mais diversas linguagens de programação, pois normalmente utilizam a sintaxe normal de chamada de funções. Existem usos bastante interessantes como veremos a seguir.

Toda função recursiva deve ter uma (ou mais) condição(ões) de parada, sem a(s) qual(is) se tornaria basicamente um *loop* infinito e pararia em um erro de estouro de pilha de chamadas (*stack overflow*), ao consumir todo o espaço dedicado para tal fim.

Em Python a exceção gerada é `RecursionError` com a mensagem *maximum recursion depth exceeded*.



Exercício 57 - Cálculo de factorial usando recursividade

fatorial_recursivo.py

```
#!/usr/bin/python3

def fatorial(n):
    return n * (fatorial(n - 1) if (n - 1) > 1 else 1) ① ②

if __name__ == '__main__':
    print(f'10! = {fatorial(10)} (6 semanas em segundos)') ③
```

① Chamada recursiva a função fatorial

② Condição de parada através do operador ternário

③ 6 semanas * 7 dias * 24 horas * 60 minutos * 60 segundos = 3628800 segundos

9.6.1. Outros exemplos de recursividade

- [Exercício 38 - Fibonacci recursivo](#)
- [Exercício 39 - Fibonacci recursivo com operador ternário](#)
- [Exemplo Avançado 1 - Fibonacci recursivo decrescente sem memoize](#)
- [Exemplo Avançado 2 - Fibonacci recursivo com decorators, memoize e classes](#)

9.7. Immutability - Imutabilidade

Imutabilidade ou a arte de não causar efeitos colaterais. Representados aqui por tipos como tuple, set, frozenset, int, str e muito mais além dos recursos de fatiamento e funções de transformação de iteráveis (como listas e tuplas), gerando um novo objeto (sem alterar o original), como: map, filter, sorted, reversed, etc.

Um objeto (ou variável de tipo primitivo) imutável tem algumas características interessantes, como:

- Redução (ou eliminação) de efeitos colaterais;
- Alta testabilidade;
- Funções puras, que permitem o uso *cache* facilmente;
- Entre outras.

Exercício 58 - Listar todos os meses do ano com 31 dias

immutability.py

```
from locale import setlocale, LC_ALL ①
from calendar import mdays, month_name ②
from functools import reduce ③

# Português do Brasil
setlocale(LC_ALL, 'pt_BR.utf8') ④

# Listar todos os meses do ano com 31 dias
# Funcional bem formatado
print(
    reduce( ⑤
        lambda output, nome_mes: f'{output}\n- {nome_mes}', ⑥
        map( ⑦
            lambda mes: month_name[mes], ⑧
            filter( ⑨
                lambda mes: mdays[mes] == 31, ⑩
                range(1, len(month_name)) ⑪
            )
        ),
        'Meses com 31 dias:', ⑫
    )
)
```

① TODO:

② TODO:

③ TODO:

④ TODO:

⑤ TODO:

⑥ TODO:

⑦ TODO:

⑧ TODO:

⑨ TODO:

⑩ TODO:

⑪ TODO:

⑫ TODO:

Exercício 59 - Listar todos os meses do ano com 31 dias (funcional em uma linha)

immutability_oneline.py

```
from locale import setlocale, LC_ALL
from calendar import mdays, month_name
from functools import reduce

# Português do Brasil
setlocale(LC_ALL, 'pt_BR.utf8')

# Listar todos os meses do ano com 31 dias
# Funcional em uma linha
print(reduce(lambda output, nome_mes: f'{output}\n- {nome_mes}', map(lambda mes: month_name[mes], filter(lambda mes: mdays[mes] == 31, range(1, len(month_name)))), 'Meses com 31 dias:'))
```

Exercício 60 - Listar todos os meses do ano com 31 dias (imperativo)

imperativo.py

```
from locale import setlocale, LC_ALL ①
from calendar import mdays, month_name ②

# Português do Brasil
setlocale(LC_ALL, 'pt_BR.utf8') ④

# Listar todos os meses do ano com 31 dias
# Equivalente no imperativo
print('Meses com 31 dias:')
for mes in range(1, len(month_name)):
    if mdays[mes] == 31:
        print(f'- {month_name[mes]}')
```

Exercício 61 - Diversas funções úteis trabalham com objetos imutáveis

immutability_functions.py

```
from functools import reduce
from operator import add ①

valores = (30, 10, 25, 70, 100, 94) ②

print(sorted(valores)) ③
print(min(valores)) ④
print(max(valores)) ⑤
print(sum(valores)) ⑥
print(reduce(add, valores)) ⑥ ⑦

print(reversed(valores)) ⑧
print(tuple(reversed(valores))) ⑨
```

① TODO: Uso do modulo operator: <https://docs.python.org/3/library/operator.html>

② TODO:

③ TODO:

④ TODO:

⑤ TODO:

⑥ TODO:

⑦ TODO:

⑧ TODO:

⑨ TODO:

9.8. Lazy Evaluation - Avaliação preguiçosa

Recurso que atrasa o máximo possível um determinado processamento, fazendo-o somente quando o mesmo for absolutamente necessário, é o inverso de *eager evaluation*.

Em Python, desde a versão 2.0, diversas funções da biblioteca padrão começaram a ser convertidas para *lazy*, um exemplo clássico foi o `xrange`, que conviveu até a versão 3 com o `range` original. A diferença é que o `range` original retornava um *list*, totalmente gerada, consumindo todo processamento e memória de uma vez, enquanto o `xrange` retornava uma espécie de *generator*, que gerava os valores sob-demanda, gastando menos CPU e consumindo muito menos memória, pode parecer irrelevante em um `range(10)`, mas se fosse um `range(16777216)`?

Na versão 3 elas foram unificadas em uma só que retorna um objeto do tipo `range`, que é um *generator*, se for necessário é possível aplicar um *eager evaluation* transformando-o em outro objeto, como uma lista ou tupla: `list(range(10))`. Isso só foi possível pois toda a linguagem foi ajustada para tornar o uso de *generators* o mais transparente possível.

Entre as funções que geram *generators*, temos:

- map

- filter
- reversed
- range

9.8.1. Generators

Para utilizarmos *lazy evaluation* em nosso próprio código temos o recurso de *generators*, que nada mais é do que uma função que possui retornos parciais, de uma iteração, e que podem ser iteradas através da função `next` do `__builtins__` até ser totalmente consumida, quando levantam uma exceção, ainda é possível consumir através de qualquer construção em Python que trabalhe com iteráveis (como no `for`, *list comprehension*, etc), e neste caso o tratamento da exceção é automático.

Exercício 62 - Consumindo generators com while e uso do yield

`generators_v1.py`

```
#!/usr/bin/python3

def cores_arco_iris():
    yield 'vermelho' ①
    yield 'laranja'
    yield 'amarelo'
    yield 'verde'
    yield 'azul'
    yield 'índigo'
    yield 'violeta'

if __name__ == '__main__':
    generator = cores_arco_iris()
    print(type(generator)) ②

    while True: ③
        print(next(generator)) ④
```

① TODO:

② TODO:

③ TODO:

④ TODO:

Exercício 63 - Consumindo generators com for

generators_v2.py

```
#!/usr/bin/python3
from generators_v1 import cores_arco_iris

if __name__ == '__main__':
    generator = cores_arco_iris()

    for cor in generator: ① ②
        print(cor)
```

① TODO:

② TODO:

Exercício 64 - Implementação do generator map

map.py

```
#!/usr/bin/python3
from functools import reduce
from first_class_functions import dobro

# Implementação simplificada do map
def mapear(function, lista): ①
    for elemento in lista:
        yield function(elemento) ②

if __name__ == '__main__':
    print(
        reduce(
            lambda output, linha: output + '\n' + linha,
            mapear(③
                lambda tupla: f'{tupla[0]} x 2 = {tupla[1]}',
                mapear(
                    lambda valor: (valor, dobro(valor)),
                    range(1, 11)
                )
            )
        )
    )
```

① TODO:

② TODO:

③ TODO:

9.8.2. Generator Expression

Exercício 65 - Implementação do map com generator expression

map_generator_expression.py

```
#!/usr/bin/python3
from functools import reduce
from first_class_functions import dobro

# Implementação simplificada do map
def mapear(function, lista):
    return (function(elemento) for elemento in lista) ①

if __name__ == '__main__':
    print(
        reduce(
            lambda output, linha: output + '\n' + linha,
            mapear(
                lambda tupla: f'{tupla[0]} x 2 = {tupla[1]}',
                mapear(
                    lambda valor: (valor, dobro(valor)),
                    range(1, 11)
                )
            )
        )
    )
```

① TODO:

9.9. Nem tudo são flores... ☺

Como já discutido antes, Python possui capacidades muito encontradas em linguagens funcionais, mas não é uma especialidade da linguagem, segue abaixo algumas capacidades que não estão facilmente disponíveis na linguagem:

- *Pattern Matching*: Regras que definem a função exata que será chamada (normalmente funções com o mesmo nome) conforme um conjunto de padrões definidos, padrões esses que não incluem apenas o tipo, incluindo valores, faixas, números de parâmetros, etc;
- *Tail Call Optimization*: Otimização de chamada recursivas, permitindo maior performance e economia de recursos;
- *Composition*: Criação de uma sequência de funções a ser executada para cada elemento da iteração, apesar de ser possível simular o recurso, não existe um suporte explícito.



Existem bibliotecas que aumentam as capacidades funcionais da linguagem, mas que estão fora do escopo deste curso.

Habilidades adquiridas ☺

- Instruções:

`yield`

Retornos parciais de *generators*;

- Biblioteca padrão (*baterias incluídas*):

`calendar`

Manipulação de calendário;

`locale`

Localização e internacionalização;

`operator`

Funções otimizadas (em C no caso do CPython) para as mais diversas operações matemáticas, lógicas, relacionais, etc;

`functools`

Diversas funções para auxiliar programação funcional;

- Funções:

`reduce`

Processar um iterável transformando-o em um valor final, na versão 3 ela saiu do `__builtins__` e foi para o módulo `functools` da biblioteca padrão;

`zip`

Combinar dois iteráveis em um;

`filter`

Criar uma nova lista a partir de outra lista com um determinado filtro;

`map`

Criar uma nova lista transformando cada um dos elementos, através de uma função;

`sorted`

Criar uma nova lista ordenada;

`reversed`

Criar uma nova lista invertida;

`max`

Itera um objeto retornando o de maior valor (podendo receber uma função para ajustar a lógica de comparação);

`min`

Idem ao `max`, porém retornando o mínimo;

`sum`

Itera um objeto somando seus elementos (podendo receber uma função para ajustar a lógica de acúmulo);

`next`

Retorna o valor da próxima iteração em um generator

- `function.__name__` propriedade de um objeto do tipo função contendo o nome da mesma;
- Mais exemplos práticos de recursividade;

- Criação de funções anônimas com o `lambda`
- *Generators e Generator Expression*
- *Unpacking* automático no `for`.

Alguns links relevantes na documentação oficial para aumentar o leque de possibilidade de Python como linguagem funcional:



- [Functional Programming HOWTO](#)
- [Functional Programming Modules](#)
- [Higher-order functions and operations on callable objects](#)

Desafio

Escrever uma função para calcular o MDC (*Máximo Divisor Comum*) de uma lista de inteiros. Sugerimos o seguinte algoritmo:

1. Escolher o menor número da lista
2. Calcular o resto da divisão de cada um dos números da lista por este número
3. Caso todos os restos sejam 0, este é o MDC
4. Senão subtrair um e voltar ao passo 2

O MDC sempre será encontrado, nem que seja o número 1.

Resultados esperados

```
if __name__ == '__main__':
    print(mdc([21, 7])) # 7
    print(mdc([125, 40])) # 5
    print(mdc([9, 564, 66, 3])) # 3
    print(mdc([55, 22])) # 11
    print(mdc([15, 150])) # 15
    print(mdc([7, 9])) # 1
```

Exemplo de solução disponível em [Desafio 5 - MDC](#)

Quero mais

Solução extremamente “funcional” para o cálculo do MDC:

- [Exemplo Avançado 4 - MDC Funcional](#)

Existem ainda mais exercícios avançadas sobre programação funcional nos anexos:

- [Exemplo Avançado 5 - Fatorial](#)
- [Exemplo Avançado 6 - Torre de Hanoi](#)

10. Às funções e além!

Assim como objetos, funções são construções extremamente poderosas, inter-relacionadas (*toda função é um objeto e todo objeto pode ter comportamento de função*) e importantes em Python. Apesar do capítulo anterior ([Programação funcional](#)) explorar bastante o uso delas, este uso tinha um determinado foco, que ao mesmo tempo significou conhecer diversas propriedades sobre as mesmas, mas deixou diversas outras de fora, por não estarem diretamente alinhadas ao paradigma funcional.

Com isso, vamos ir além e nos tornar mais fluentes neste recurso.



Sim, parafraseei **Buzz Lightyear!** 😊

10.1. Tipos de parâmetros

Em Python temos basicamente dois tipos de parâmetros:

Parâmetro posicional

A posição do parâmetro da lista determina a ordem dos argumentos, todos os posicionais são obrigatórios, menos o especial (*star arg*) que utiliza *unpacking* para receber todo o excesso de argumentos posicionais

Parâmetro nomeado

A associação entre o argumento e o parâmetro ocorre através do nome, porém excesso de argumentos posicionais (em relação aos parâmetros definidos) podem ser atribuídos aos parâmetros nomeados na ordem em que aparecem (esquerda para direita) ou até encontrar o parâmetro especial posicional (*star arg*) que é precedido de um asterisco. Os nomeados também possuem um especial que “pega” qualquer excesso de argumentos nomeados que é precedido de dois asteriscos. Os parâmetros nomeados devem ter um valor *default*.



Os parâmetros especiais normalmente são chamados de `*args` e `**kwargs`, sendo dos tipos `tuple` e `dict` respectivamente.

Parâmetro × Argumento



Quando usamos parâmetro nos referimos à variável que receberá o valor passado pela chamada da função, enquanto argumento é exatamente o valor passado.

Fonte: [Wikipedia](#)

10.2. Parâmetros nomeados ou opcionais

Os parâmetros opcionais são extremamente úteis para permitir uma maior flexibilidade na função, assumindo comportamentos padrões (convenção sobre configuração) e diminuindo a API obrigatória da mesma.

Isso é feito através da definição de valores padrões (*default*) nos parâmetros, porém há certas

regras:

- Os parâmetros são processados da esquerda para direita, como existem os parâmetros especiais que “pegam” vários argumentos, o ordem dos parâmetros (mesmo opcionais) é extremamente relevante, afetando o resultado final, o ideal é que todos os parâmetros opcionais venham após os parâmetros posicionais;
- Valores default podem ser expressões (mas são avaliadas no mesmo momento da definição da função no *namespace*);
- ☠ Requer muito cuidado ao usar tipos mutáveis.

Exercício 66 - Parâmetros opcionais (com valores *default*)

html_generator_v1.py

```
#!/usr/bin/python3

def build_block(texto, classe='success'): ①
    return f'<div class="{classe}">{texto}</div>'

if __name__ == '__main__':
    # Testes (assertions) ②
    assert build_block('Incluído com sucesso!') == '<div class="success">Incluído com sucesso!</div>'
    assert build_block('Impossível excluir!', 'error') == '<div class="error">Impossível excluir!</div>'

    print(build_block('ok'))
```

① Definição da classe CSS a ser utilizada quando não for especificada alguma mais específica

② Uso da instrução `assert` para testes simples, caso o teste falhe é levantado uma exceção: `AssertionError`

10.3. Argumentos nomeados

Exercício 67 - Argumentos nomeados

html_generator_v2.py

```
#!/usr/bin/python3

def build_block(texto, classe='success', inline=False): ①
    tag = 'span' if inline else 'div' ②
    return f'{tag} {classe}>{texto}</{tag}<'

if __name__ == '__main__':
    print(build_block('bloco'))
    print(build_block('linha e classe', 'info', True)) ③
    print(build_block('linha', inline=True)) ④
    print(build_block('falhou', classe='error')) ④
```

- ① TODO
- ② TODO
- ③ TODO
- ④ TODO

10.4. Unpacking de argumentos

Exercício 68 - Unpacking de argumentos

html_generator_v3.py

```
#!/usr/bin/python3

def build_block(texto, classe='success', inline=False):
    tag = 'span' if inline else 'div'
    return f'{tag} {classe}>{texto}</{tag}<'

def build_list(*itens): ①
    lista = ''.join(f'{item}</li>' for item in itens)
    return f'{ul}{lista}{ul}<'

if __name__ == '__main__':
    print(build_block('bloco'))
    print(build_block('linha e classe', 'info', True))
    print(build_block('linha', inline=True))
    print(build_block('falhou', classe='error'))
    print(build_block(build_list('Sábado', 'Domingo'), classe='info')) ②
```

- ① TODO
- ② TODO

10.5. Combinando *unpacking* e parâmetros opcionais

Exercício 69 - Combinando *unpacking* e parâmetros opcionais

html_generator_v4.py

```
#!/usr/bin/python3

def build_block(conteudo, *args, classe='success', inline=False): ①
    tag = 'span' if inline else 'div'
    html = conteudo if not callable(conteudo) else conteudo(*args) ②
    return f'{tag} class="{classe}">{html}</{tag}>'

def build_list(*itens):
    lista = ''.join(f'<li>{item}</li>' for item in itens)
    return f'<ul>{lista}</ul>'

if __name__ == '__main__':
    print(build_block('bloco'))
    print(build_block('linha e classe', 'info', True)) ③
    print(build_block('linha', inline=True))
    print(build_block('falhou', classe='error'))
    print(build_block(build_list('Sábado', 'Domingo'), classe='info'))
    print(build_block(build_list, 'Sábado', 'Domingo', classe='info'))
```

① TODO

② TODO

③ TODO



Avaliar o que ocorre se o parâmetro especial (*star arg*) vier antes dos parâmetros nomeados

10.6. *Unpacking* de argumentos nomeados

Exercício 70 - Unpacking de argumentos nomeados

html_generator_v5.py

```
#!/usr/bin/python3

block_attributes = ('accesskey',) ①
li_attributes = ('type',) ①

def filteredAttrs(kwargs, filter): ②
    return ''.join(f'{k}="{v}"' for k, v in kwargs.items() if k in filter)

def build_block(conteudo, *args, classe='success', inline=False, **kwargs): ③
    tag = 'span' if inline else 'div'
    html = conteudo if not callable(conteudo) else conteudo(*args, **kwargs) ④
    atributos = {'class': classe} ⑤
    atributos.update(kwargs)
    return f'<{tag} {filteredAttrs(atributos, block_attributes + ("class",))}>{html}</{tag}>'

def build_list(*itens, **kwargs): ⑥
    lista = ''.join(f'<li {filteredAttrs(kwargs, li_attributes)}>{item}</li>' for item in itens)
    return f'<ul>{lista}</ul>'

if __name__ == '__main__':
    print(build_block('bloco'))
    print(build_block('linha', inline=True))
    print(build_block('falhou', classe='error'))
    print(build_block(build_list('Sábado', 'Domingo'), classe='info'))
    print(build_block(build_list, 'Sábado', 'Domingo', classe='info'))
    print(build_block(build_list, 'Sábado', 'Domingo', classe='info', accesskey='m', type='square')) ⑦
```

① TODO

② TODO

③ TODO

④ TODO

⑤ TODO

⑥ TODO

⑦ TODO

10.7. Objetos chamáveis

Exercício 71 - Objetos chamáveis

callable_object.py

```
#!/usr/bin/python3

class ClosureClass(object): ①
    """Calcula uma potência específica"""\n\n    def __init__(self, potencia): ③
        self.potencia = potencia\n\n    def __call__(self, valor): ④
        return valor ** self.potencia\n\n\nif __name__ == '__main__':
    quadrado = ClosureClass(2) ⑤
    cubo = ClosureClass(3)\n\n    if callable(quadrado):
        print('quadrado: Objetos desta classe podem atuar como função')\n\n    print(f'Documentação: {ClosureClass.__doc__}')
    print(f'3² => {quadrado(3)})')
    print(f'5³ => {cubo(5)})')
```

- ① Veremos classes com mais detalhes em [Programação orientada a objetos](#)
- ② Documentação da classe: `__doc__`
- ③ Constructor da classe, que recebe a potência
- ④ Implementação do método especial `__call__` indica que o objeto poderá ser chamado como uma função
- ⑤ Os objetos resultantes funcionarão como funções com uma *closure* (já que retém o estado da construção da classe)

10.8. Problemas com argumentos mutáveis

Exercício 72 - Problemas com argumentos mutáveis 💀

mutable_default_argument.py

```
#!/usr/bin/python3

def fibonacci(sequencia=[0, 1]): ①
    """Uso de mutáveis como valor default (armadilha)"""

    sequencia.append(sequencia[-1] + sequencia[-2])
    return sequencia

if __name__ == '__main__':
    inicio = fibonacci()
    print(inicio, id(inicio)) ②
    print(fibonacci(inicio))

    restart = fibonacci()
    print(restart, id(restart)) ③
    assert restart == [0, 1, 1]
```

① Caso um argumento não seja passado é assumido: [0, 1]

② Imprime o inicio da sequência e o seu endereço de memória

③ Imprime a **tentativa** de pegar de novo o inicio da sequência e o seu endereço de memória

Isso acontece por que a execução de todas as expressões na assinatura da função ocorrem apenas uma vez, e o resultado de qualquer expressão é armazenado junto com a função e reaproveitado toda vez que o argumento não é passado.



Esta característica pode ser usada também como uma *feature*, algo similar ao que conseguimos com *closure*.

Exercício 73 - Problemas com argumentos mutáveis (solução)

mutable_default_argument_fix.py

```
def fibonacci(sequencia=None): ①
    sequencia = sequencia or [0, 1] ②
    sequencia.append(sequencia[-1] + sequencia[-2])
    return sequencia
```

① Uso de um argumento *default* imutável

② Substituição condicional pelo valor mutável

10.9. Decorators

Exercício 74 - Decorator log

decorator.py

```
#!/usr/bin/python3

def log(function): ①
    def decorator(*args, **kwargs): ②
        print(f'Inicio da chamada da função: {function.__name__}')
        print(f'args: {args}')
        print(f'kwargs: {kwargs}')
        resultado = function(*args, **kwargs) ③
        print(f'Resultado da chamada: {resultado}')
        return resultado
    return decorator ④

@log ⑤
def soma(x, y):
    return x + y

@log ⑤
def sub(x, y):
    return x - y

if __name__ == '__main__':
    print(soma(5, 7))
    print(sub(5, y=7))
```

① Função log, que age como um *decorator*

② *Closure* para execução do *decorator* ao mesmo tempo que guarda a referência da função original que foi passada como argumento para o *decorator*

③ Chamada da função original, com todos os parâmetros

④ Retorno da função interna, que possui o código do *decorator* e a chamada da função original

⑤ Uso do *decorator* nas funções (*soma* e *sub*), com a sintaxe: `@decorator` uma linha antes da instrução da função



A linguagem possui alguns *decorators* na biblioteca padrão, veremos alguns no capítulo de orientação a objetos como `classmethod` e `staticmethod`.

Habilidades adquiridas

- Instruções:

`assert`
Avaliação simples de expressões para fins de testes;

- Funções:

`id`

Retorna o identificador de um objeto, em **CPython**, este identificador é o endereço de memória

- Exceções:

`AssertionError`

Exceção levantada quando um `assert` falha;

- Métodos e propriedades especiais (ou mágicos):

`__call__`

Qualquer objeto que implemente este método, pode ser chamado com a sintaxe de função;

`__doc__`

Uma propriedade que permite a definição de uma *string* como documentação da função;

- Parâmetros opcionais (com valores *default*);
- *Unpacking* de parâmetros (tupla de parâmetros);
- *Unpacking* de parâmetros nomeados (dicionário de parâmetros);
- Parâmetros *default* e o problema com objetos mutáveis;
- *Decorators*.

Desafio

Criar uma função que retorne HTML genérico, abrindo e fechando as *tags*, suportando quaisquer atributos na *tag* e o seu conteúdo pode ser um texto ou uma lista com vários textos (ou outras *tags* já como texto).

Caso encontre um atributo chamado de `css`, renomeá-lo para `class`. Isso é necessário por que `class` é uma palavra reservada em Python, e não poderia ser usado como literal na chamada da função.

Exemplo de chamada da nova função tag

```
tag('p',
    tag('span', 'Curso de Python 3, por '),
    tag('strong', 'Juracy Filho', id='jf'),
    tag('span', ' e '),
    tag('strong', 'Leonardo Leitão', id='ll'),
    tag('span', '.'),
    css='alert')
```

Retorno esperado

```
<p class="alert"><span >Curso de Python 3, por </span><strong id="jf">Juracy Filho</strong><span > e </span><strong id="ll">Leonardo Leitão</strong><span >.</span></p>
```



A provável assinatura da função seria: `tag(tag, *args, **kwargs)`

Exemplo de solução disponível em [Desafio 6 - Gerador de HTML](#)

Quero mais

Mais informações disponíveis na comunidade:

- [Common Gotchas - Mutable Default Arguments](#)

Exemplo avançado usando o recurso de *decorator* através de uma classe:

- [Exemplo Avançado 2 - Fibonacci recursivo com decorators, memoize e classes](#)

11. Dominando as instruções Python

Como vimos até o momento, a linguagem possui diversas instruções para executar laços, desvios de fluxo e outras operações básicas. Vimos várias deles em [Instruções](#).

Utilizamos no primeiro momento as formas mais simples dessas instruções, que são muito parecidas com outras linguagens, porém o pulo do gato está nas opções menos triviais, como veremos neste capítulo.



Alguns links relevantes sobre as instruções Python:

- [Compound statements](#)

11.1. E se... senão se...

A instrução `if`, além de poder ter uma cláusula `else`, pode ter vários `elif's`, sim uma forma compacta (e bastante estranha) da concatenação de `else` mais `if`. Uma mesma instrução `if` pode ter 0 ou vários `elif's`.

Exercício 75 - Conhecendo o `elif`

`if_statement.py`

```
def check_idade(idade):
    if 0 < idade < 18:
        return 'Menor de idade'
    elif idade in range(18, 50): ①
        return 'Adulto'
    elif idade in range(51, 100): ①
        return 'Melhor idade'
    elif idade >= 100: ①
        return 'Centenário'
    else: # Provavelmente negativo
        return 'idade inválida'

if __name__ == '__main__':
    for idade in (17, 35, 87, 113, -2):
        print(f'{idade}: {check_idade(idade)})
```

① A cláusula `elif` avalia uma expressão do mesmo modo que o `if` principal.



Apenas um bloco é executado, seja o `if`, `elif` ou `else`.



Evite o uso de vários `elif's`, existem formas mais pythônicas para isso.

11.2. *Switch? Case? Não, obrigado!*

Não existe uma instrução dedicada para tratar expressões baseadas em uma variável/expressão inicial. Porém existem alternativas bastante atrativas utilizando recursos da própria linguagem.

Exercício 76 - Simulando um switch

switch_statement_v1.py

```
def get_day_name(dia):
    dias = {
        1: 'Domingo',
        2: 'Segunda',
        3: 'Terça',
        4: 'Quarta',
        5: 'Quinta',
        6: 'Sexta',
        7: 'Sábado',
    }

    return dias.get(dia, '** inválido **') ①

if __name__ == '__main__':
    for dia in range(0, 9):
        print(f'{dia}: {get_day_name(dia)})'
```

- ① O método `get` do `dict` possui um parâmetro opcional `default`, que permite indicar um valor quando a chave não for encontrada.

Exercício 77 - Switch com valores únicos

switch_statement_v2.py

```
def get_day_type(dia):
    dias = {
        1: 'Fim de semana',
        2: 'Dia de semana',
        3: 'Dia de semana',
        4: 'Dia de semana',
        5: 'Dia de semana',
        6: 'Dia de semana',
        7: 'Fim de semana',
    }

    return dias.get(dia, '** inválido **')

if __name__ == '__main__':
    for dia in range(0, 9):
        print(f'{dia}: {get_day_type(dia)})'
```

Exercício 78 - Switch baseado em faixa de valores

switch_statement_v3.py

```
def get_day_type(dia):
    dias = {
        (1, 7): 'Fim de semana', ①
        tuple(range(2, 7)): 'Dia de semana', ①
    }

    dict_in_key = (msg for k, msg in dias.items() if dia in k) ②
    return next(dict_in_key, '** dia inválido **') ③

if __name__ == '__main__':
    for dia in range(0, 9):
        print(f'{dia}: {get_day_type(dia)})
```

- ① Chave do dicionário deve ser um imutável (regra do `dict`) e suportar o operador `in` para atender a lógica de *lookup*
- ② *Expression generator* para fazer o `get` usando o operador `in` em vez de igualdade
- ③ A função `next` suporta um segundo parâmetro que é um valor `default` caso o `iterator` não gere um próximo valor: `StopIteration`

11.3. Laços condicionais *like a boss*

A instrução `while`, além de suportar a instrução `break`, suporta também a instrução `continue` e `else` ... sim isso mesmo: `else`.

Exercício 79 - Conhecendo a fundo o while

while_statement.py

```
from random import randint

numeros = []

# Gerar números randômicos pares, até o limite de 10
# Com parada imediata se o último número randômico coincidir com a quantidade
while len(numeros) < 10:
    numero = randint(1, 20) ①

    if numero % 2 == 1: ②
        continue ③

    numeros.append(numero)

    if numero == len(numeros):
        print('BINGO 🎉', numeros)
        break ④
    else:
        print(numeros) ⑤
```

- ① A função randint do módulo random, gera um número randômico na faixa indicada
- ② Avalia se o número é ímpar
- ③ Instrução continue pula toda a lógica do bloco e a condição do laço é reavaliada
- ④ Instrução break encerra imediatamente o laço, não executando a cláusula else caso exista
- ⑤ A cláusula else do while é executada **apenas** se não ocorrer um break

11.4. Iterações *like a boss*

A instrução for, assim como o while suporta as instruções break, continue e else. Possuindo a mesma lógica.

Exercício 80 - Conhecendo a fundo o `for`

for_statement.py

```
from random import randint

def d100():
    """Dados de 100 lados"""
    return randint(0, 100)

for i in range(100):
    if i % 2 == 1: ①
        continue

    if d100() == i: ②
        print('BINGO 🎉', i)
        break
else: ③
    print('Não tivemos um vencedor! 😞')
```

① Pular os números ímpares

② Caso o valor do `d100` seja igual ao número da iteração atual... **Bingo** e encerra a iteração

③ Caso não tenha ocorrido um `break` finaliza sem sucesso!

Exercício 81 - Uso de variável de controle extra no `for`

for_sem_else.py

```
PALAVRAS_PROIBIDOS = ('futebol', 'religião', 'politica')

textos = [
    'João gosta de futebol e politica',
    'A praia foi divertida',
]

for texto in textos:
    found = False ①
    for palavra in texto.lower().split():
        if palavra in PALAVRAS_PROIBIDOS: ②
            print('Texto possui ao menos uma palavra proibida:', palavra)
            found = True
            break

    if not found: ③
        print('Texto autorizado:', texto)
```

① Variável de controle que indica se encontrou alguma palavra proibida, começa com `False`

② Caso alguma das palavras seja encontrada, ativa a variável de controle, indica que palavra achou e encerra a iteração

③ Caso nenhuma palavra seja encontrada (infelizmente só é possível aferir isso após concluir a iteração), autoriza o texto

Exercício 82 - Uso do else no for

for_com_else.py

```
PALAVRAS_PROIBIDOS = ('futebol', 'religião', 'politica')

textos = [
    'João gosta de futebol e politica',
    'A praia foi divertida',
]

for texto in textos:
    for palavra in texto.lower().split():
        if palavra in PALAVRAS_PROIBIDOS: ①
            print('Texto possui ao menos uma palavra proibida:', palavra)
            break
    else: ②
        print('Texto autorizado:', texto)
```

① Caso alguma das palavras seja encontrada, indica que palavra achou e encerra a iteração

② Caso a iteração não tenha sido encerrada com um `break`, autoriza o texto



Esta última solução possui não necessita de variável de controle para saber se o `for` concluiu normalmente ou foi encerrada por um `break`.

Desafio (extra)

Escrever o mesmo exercício das palavras proibidas que vimos em [Exercício 81 - Uso de variável de controle extra no for](#) e [Exercício 82 - Uso do else no for](#) utilizando set.

O set possui um método especial chamado `intersection` que recebe um outro set e retorna um set com a interseção encontrada.

Exemplo de solução disponível em [Desafio 7 - Palavras proibidas com set](#)

11.5. Tratamento de exceções *like a boss*

Até o momento vimos duas variações do `try`, um para realmente tratar a exceção: `try...except` e um outro para código de encerramento (com ou sem exceção): `try...finally`. O que nós não vimos foi que não só eles podem ser combinados como ainda existe uma cláusula `else` para um bloco de código que só será executado se não ocorrer nenhuma exceção.

Exercício 83 - Conhecendo a fundo o try

try_statement.py

```
try:
    print('try (sem provocar exceções)')
except Exception as e:
    print('except', e)
else:
    print('else')
finally:
    print('finally')

print('*' * 20)

try:
    print('try (provocando exceções)')
    print('divisão:', 2 / 0)
except Exception as e:
    print('except', e)
else:
    print('else')
finally:
    print('finally')

print('*' * 20)

# Sequência completa sem ocorrer exceção e sem else
try:
    print('try (sem else)')
except Exception as e:
    print('except', e)
finally:
    print('finally')
```

Habilidades adquiridas

- Instruções:

else

Compondo outras instruções: for, while e try;

continue

Compondo outras instruções: for e while;

try

Formas mais completas para tratamento de exceções.

12. Packages

Exercício 84 - Execução de uma função em outro package

package1/__init__.py



A existência do arquivo `__init__.py` (mesmo vazio) serve para indicar que o diretório `package1` é um pacote **Python**.

package1/modulo1.py

```
print('importado')

def soma(x, y):
    return x + y
```

package_v1.py

```
from package1 import modulo1

print(type(modulo1))
print(modulo1.soma(2, 3))
```

Exercício 85 - Momento de execução do código

package1/modulo2.py

```
def main():
    print('Rodando o main()')

if __name__ == '__main__':
    main()
```

package_v2.py

```
from package1 import modulo2

print('O main só será executado através de uma chamada explícita')
modulo2.main()
```

Exercício 86 - Uso de módulos com mesmo nome

package2/__init__.py

```
def subtracao(x, y):
    return x - y
```

package_v3.py

```
from package1 import modulo1
from package2 import modulo1 as modulo1_sub

print('Soma', modulo1.soma(3, 2))
print('Subtração', modulo1_sub.subtracao(3, 2))
```

Exercício 87 - Importação direta das funções no *namespace* atual

package_v4.py

```
from package1.modulo1 import soma
from package2.modulo1 import subtracao

print('Soma', soma(3, 2))
print('Subtração', subtracao(3, 2))
```

Exercício 88 - Uso de um pacote como *façade*

calc/__init__.py

```
from package1.modulo1 import soma
from package2.modulo1 import subtracao

__all__ = ['soma', 'subtracao']
```

package_v5.py

```
from calc import soma, subtracao

print('Soma', soma(3, 2))
print('Subtração', subtracao(3, 2))
```

Habilidades adquiridas

Criação de pacotes para segmentar e estruturar melhor o funcionamento de uma aplicação ou biblioteca. E com isso mais algumas funcionalidade.

- Uso do `__all__` para importação total de identificadores;
- Importação de identificadores específicos;
- Renomear identificadores na importação, particularmente.

Desafio 🎯

Criar pacotes com funções que permitam o funcionamento do código abaixo:

Exercício 89 - Consumidor do pacote do desafio

`desafio_package.py`

```
from app.utils.generators import nome_proprio
from app.negocio import check_exists
from app.negocio.backend import add_nome

def main():
    while True:
        nome = nome_proprio()
        if not check_exists(nome):
            add_nome(nome)
            break

    print(f'Criado novo nome de testes: "{nome}"')

if __name__ == '__main__':
    main()
```

Exemplo de solução disponível em [Desafio 8 - Pacote](#) app.

13. Programação orientada a objetos

13.1. Classe Task

Exercício 90 - Classe Task

todo_v1.py

```
#!/usr/bin/python3
from datetime import datetime

class Task(object):
    def __init__(self, descricao):
        self.descricao = descricao
        self.feito = False
        self.criacao = datetime.now()

    def done(self):
        self.feito = True

    def __str__(self):
        return f'{self.descricao}{' (feito)' if self.feito else ''}

def main():
    casa = []
    casa.append(Task('Passar roupa'))
    casa.append(Task('Lavar prato'))
    [task.done() for task in casa if task.descricao == 'Lavar prato']
    for task in casa:
        print(f'- {task}')

if __name__ == '__main__':
    main()
```

13.2. Classe Project

Exercício 91 - Classe Project

todo_v2.py

```
class Project(object):
    def __init__(self, nome):
        self.nome = nome
        self.tasks = []

    def add(self, descricao):
        self.tasks.append(Task(descricao))

    def pendentes(self):
        return [task for task in self.tasks if not task.feito]

    def find(self, descricao):
        # Possível IndexError
        return [task for task in self.tasks if task.descricao == descricao][0]

    def __str__(self):
        return f'{self.nome} ({len(self.pendentes())} tarefas pendentes)'

def main():
    casa = Project('Casa')
    casa.add('Passar roupa')
    casa.add('Lavar prato')

    mercado = Project('Compras no mercado')
    mercado.add('Frutas secas')
    mercado.add('Carne')
    mercado.add('Tomate')

    casa.find('Lavar prato').done()
    print(casa)
    for task in casa.tasks:
        print(f'- {task}')

    print(mercado)
    for task in mercado.tasks:
        print(f'- {task}')
```

13.3. Método __iter__

Exercício 92 - Método `__iter__`

todo_v3.py

```
class Project(object):
    def __iter__(self):
        return self.tasks.__iter__()

def main():
    casa = Project('Casa')
    casa.add('Passar roupa')
    casa.add('Lavar prato')

    mercado = Project('Compras no mercado')
    mercado.add('Frutas secas')
    mercado.add('Carne')
    mercado.add('Tomate')

    casa.find('Lavar prato').done()
    print(casa)
    for task in casa:
        print(f'- {task}')

    print(mercado)
    for task in mercado:
        print(f'- {task}')
```

Duck typing

Quando eu vejo um pássaro que anda como um pato, nada como um pato e grasna como um pato, eu o chamo de pato.

— James Whitcomb Riley

13.4. Implementação do vencimento

Exercício 93 - Implementação do vencimento (datetime e timedelta)

todo_v4.py

```
class Task(object):
    def __init__(self, descricao, vencimento=None):
        self.descricao = descricao
        self.feito = False
        self.criacao = datetime.now()
        self.vencimento = vencimento

    def __str__(self):
        decorators = []
        if self.feito:
            decorators.append('(feito)')
        elif self.vencimento:
            if datetime.now() > self.vencimento:
                decorators.append('(vencido)')
        else:
            decorators.append(f'(vence em {(self.vencimento - datetime.now()).days} dias)')

        return f'{self.descricao} ' + ''.join(decorators)

class Project(object):
    def add(self, descricao, vencimento=None):
        self.tasks.append(Task(descricao, vencimento))
```

13.5. Herança

Exercício 94 - Herança

todo_v5.py

```
class TaskRecurring(Task):
    def __init__(self, descricao, vencimento, dias=7):
        super().__init__(descricao, vencimento)
        self.dias = dias

    def done(self):
        super().done()
        return TaskRecurring(self.descricao, datetime.now() + timedelta(days=self.dias), self.dias)

def main():
    casa = Project('Casa')
    casa.add('Passar roupa')
    casa.add('Lavar prato')
    casa.add('Arrumar guarda-roupa', datetime.now() + timedelta(days=3))
    casa.add('Pintar', datetime.now() - timedelta(days=7))
    casa.tasks.append(TaskRecurring('Trocá lençóis', datetime.now(), 7))

    print(casa)
    for task in casa:
        print(f'- {task}')

    print('*** Tarefa recorrente ***')
    casa.tasks.append(casa.find('Trocá lençóis').done())
    for task in casa:
        print(f'- {task}')
```

13.6. Métodos “privados”

Exercício 95 - Métodos “privados” e simulação de “overload”

todo_v6.py

```
class Project(object):
    def __init__(self, nome):
        self.nome = nome
        self.tasks = []

    def _add_task(self, task, **kwargs):
        self.tasks.append(task)

    def _add_new_task(self, descricao, **kwargs):
        self.tasks.append(Task(descricao, kwargs.get('vencimento', None)))

    def add(self, task, vencimento=None, **kwargs):
        real_function = self._add_task if isinstance(task, Task) else self._add_new_task
        kwargs['vencimento'] = vencimento
        real_function(task, **kwargs)

def main():
    casa = Project('Casa')
    casa.add('Passar roupa')
    casa.add('Lavar prato')
    casa.add('Arrumar guarda-roupa', datetime.now() + timedelta(days=3))
    casa.add('Pintar', datetime.now() - timedelta(days=7))
    casa.add(TaskRecurring('Trocá lençóis', datetime.now(), 7))

    mercado = Project('Compras no mercado')
    mercado.add('Frutas secas')
    mercado.add('Carne')
    mercado.add('Tomate')

    casa.find('Lavar prato').done()
    print(casa)
    for task in casa:
        print(f'- {task}')

    print('*** Tarefa recorrente ***')
    casa.add(casa.find('Trocá lençóis').done())
    for task in casa:
        print(f'- {task}')
```

13.7. Sobrecarga de operador

Exercício 96 - Sobrecarga de operador

todo_v7.py

```
class TaskRecurring(Task):
    def __init__(self, descricao, vencimento, dias=7):
        super().__init__(descricao, vencimento)
        self.dias = dias
        self.parent = None

    def done(self):
        super().done()
        new_task = TaskRecurring(self.descricao, datetime.now() + timedelta(days=self.dias), self.dias)
        if self.parent:
            self.parent += new_task
        return new_task


class Project(object):
    def __iadd__(self, task):
        task.parent = self
        self._add_task(task)
        return self


def main():
    casa = Project('Casa')
    casa.add('Passar roupa')
    casa.add('Lavar prato')
    casa.add('Arrumar guarda-roupa', datetime.now() + timedelta(days=3))
    casa.add('Pintar', datetime.now() - timedelta(days=7))
    casa += TaskRecurring('Trocar lençóis', datetime.now(), 7)

    mercado = Project('Compras no mercado')
    mercado.add('Frutas secas')
    mercado.add('Carne')
    mercado.add('Tomate')

    casa.find('Lavar prato').done()
    print(casa)
    for task in casa:
        print(f'- {task}')

    print('*** Tarefa recorrente ***')
    casa.find('Trocar lençóis').done()
    for task in casa:
        print(f'- {task}')

    print(mercado)
    for task in mercado:
        print(f'- {task}')
```

13.8. Snake trap

Exceções também são classes!



Figura 1. Snake trap

Exercício 97 - Snake trap - Tratamento de exceções

todo_v8.py

```
class TaskNotFound(Exception): ①
    pass ②

class Project(object):
    def find(self, descricao):
        try: ③
            return [task for task in self.tasks if task.descricao == descricao][0]
        except IndexError as e: ④
            raise TaskNotFound(str(e)) ⑤

def main():
    casa = Project('Casa')
    casa.add('Passar roupa')
    casa.add('Lavar prato')
    casa.add('Arrumar guarda-roupa', datetime.now() + timedelta(days=3))
    casa.add('Pintar', datetime.now() - timedelta(days=7))
    casa += TaskRecurring('Trocar lençóis', datetime.now(), 7)

    try:
        casa.find('Lavar prato - ERRO').done() ⑥
    except TaskNotFound:
        pass ②

    print(casa)
    for task in casa:
        print(f'- {task}')
```

① TODO

② TODO

③ TODO

④ TODO

⑤ TODO

⑥ TODO

Habilidades adquiridas 🎓

Usando um simples sistema de tarefas a fazer, pudemos mergulhar um pouco na programação orientada a objetos em Python, conhecendo os seguintes recursos:

- Instruções:

```
class
    Criação de classes;

pass
    Simular um bloco (blocos não podem ser vazios);

try..except
    Tratamento de exceções;
```

```
raise  
    Levantar exceção;
```

- Métodos especiais (ou mágicos):

`__init__`

Construtor da classe;

`__str__`

Como converter o objeto para string;

`__iter__`

Supporte para iteração no objeto;

`__iadd__`

Sobrecarga do operador `+=`;

- Conceito e suporte do Python para *Duck Typing*;
- Tratamento simples de datas com `datetime` e `timedelta`;
- Tratamento de exceções;
- Herança e o uso do `super()`;
- Uso de métodos como variáveis;
- Métodos “privados”;
- Uso prático do `isinstance`;
- Simulação de *overload* (sobrecarga).

Alguns links relevantes na documentação oficial:



- [Métodos especiais](#)
- [Convenção de membros de classe privados](#)

Desafio

Implementar o [Diagrama de classes](#), para gerir dados básicos de venda de uma loja.

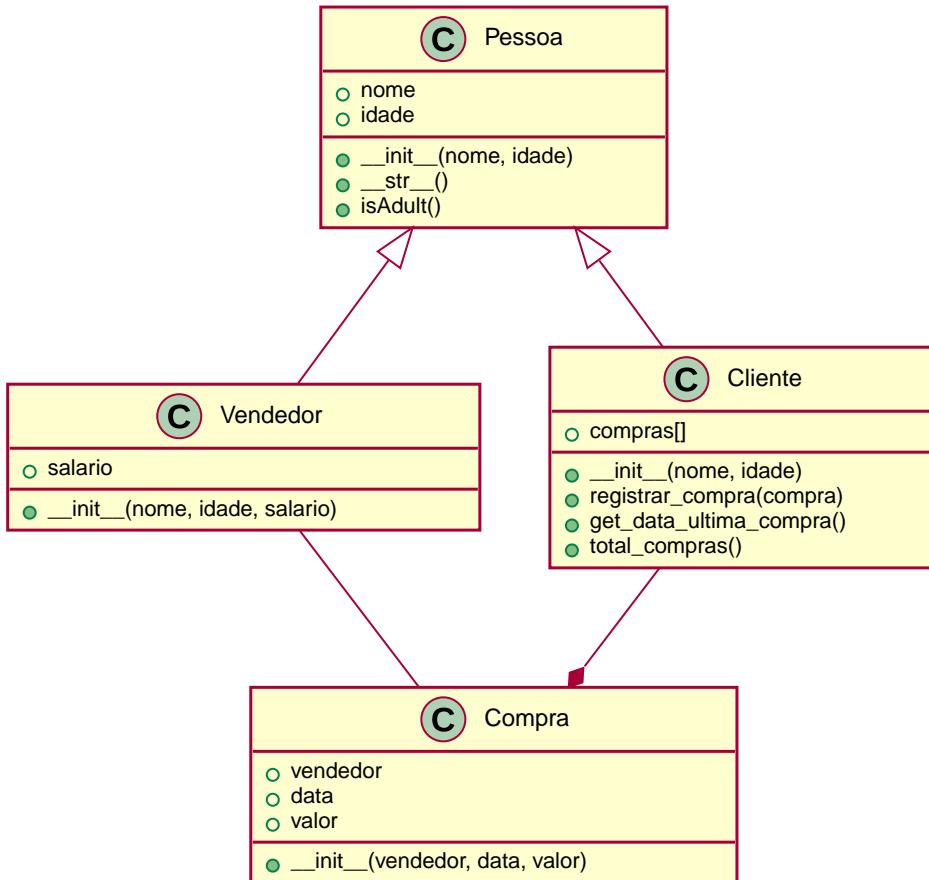


Figura 2. Diagrama de classes

Regras

- Tanto vendedor quanto cliente são pessoas (herdam da classe `Pessoa`)
- Ao converter um cliente ou vendedor em string deve mostrar o nome e a idade
- O cliente possui uma lista de compras efetuadas (do tipo `Compra`)
- O método `Cliente.registra_compra()` recebe um objeto do tipo `Compra`
- O método `Cliente.total_compras()` deve retornar o somatório de todas as compras
- O método `Cliente.get_data_ultima_compra()` deve retornar a data da última compra
- A propriedade `Compra.vendedor` é do tipo `Vendedor`

Aumento do desafio ☀️☀️

- Utilizar módulos e pacotes para melhor organização, deixando mais profissional.
- Gerir a coleção de clientes e vendedores numa classe `Loja`.

Exemplo de solução disponível em [Desafio 9 - Controle de vendas](#)

14. Orientada a objetos - Avançado

14.1. Membros de classe × membros da instância

Exercício 98 - Membros de classe × membros da instância

evolucao_v1.py

```
#!/usr/bin/python3

class EvolucaoHumana(object):
    especie = 'Homo Sapiens' ①

    def __init__(self, nome):
        self.nome = nome

    def das_cavernas(self):
        self.especie = 'Homo Neanderthalensis' ②

if __name__ == '__main__':
    jose = EvolucaoHumana('José')
    grokn = EvolucaoHumana('Grokn')
    grokn.das_cavernas()

    print(f'EvolucaoHumana.especie: {EvolucaoHumana.especie}')
    print(f'jose.especie: {jose.especie}') ③
    print(f'grokn.especie: {grokn.especie}')

    EvolucaoHumana.especie = 'Homo Sapiens Sapiens' ④
    print(f'EvolucaoHumana.especie: {EvolucaoHumana.especie}')
    print(f'jose.especie: {jose.especie}')
    print(f'grokn.especie: {grokn.especie}')
```

- ① Atributos setados dentro da classe diretamente (e não nos métodos), são membros de classe, e estão disponíveis diretamente através da classe e em todas as suas instâncias, a não ser que exista um membro de instância de mesmo nome
- ② Ao setar um atributo através do objeto/instância (`self`), o que criaremos será um membro de instância
- ③ Ao acessar um atributo em uma instância e a mesma não possuirá, uma busca é feita em sua classe (incluindo toda a herança)
- ④ Alterando o valor de um membro de classe “afeta” todas as instâncias da mesma

14.2. Métodos em profundidade

Existem 3 tipos de métodos:

- De instância
- De classe
- Estático

Até agora todos os métodos que criamos foram de instância, recebem no primeiro parâmetro a instância que disparou o método, é possível chama-lo a partir da classe mas isso exigiria passar explicitamente o `self: EvolucaoHumana.das_cavernas(pedro)`.

O método de classe utiliza o *decorator classmethod* na sua sintaxe, com isso o método passar a estar associado diretamente a classe e não a instância, porém ainda pode ser chamada a partir de um objeto. Seu primeiro parâmetro é a classe que disparou o método (que pode ser usado para polimorfismo de várias maneiras), que foi convencionado com o nome de `cls`.

O método estático é mais simples, utiliza o *decorator staticmethod* e não recebe parâmetro nenhum, pode ser chamado tanto da classe quanto da instância. Em termos práticos nada mais é do que uma função no *namespace* da classe.

Exercício 99 - Tipos de métodos

evolucao_v2.py

```
#!/usr/bin/python3

class EvolucaoHumana(object):
    especie = ''

    @staticmethod ①
    def especies():
        adjetivos = ('Habilis', 'Erectus', 'Neanderthalensis', 'Sapiens')
        return ('Australopiteco',) + tuple(f'Homo {adj}' for adj in adjetivos)

    @classmethod ②
    def is_evoluido(cls): ③
        return cls.especie == cls.especies()[-1]

    def __init__(self, nome):
        self.nome = nome

class Neanderthal(EvolucaoHumana):
    especie = EvolucaoHumana.especies()[-2] ④

class HomoSapiens(EvolucaoHumana):
    especie = EvolucaoHumana.especies()[-1] ④

if __name__ == '__main__':
    jose = HomoSapiens('José')
    grokn = Neanderthal('Grokn')

    print(f'Evolução (a partir da classe): {" ".join(HomoSapiens.especies())}') ④
    print(f'Evolução (a partir da instancia): {" ".join(jose.especies())}') ④

    print(f'Homo Sapiens evoluído? {HomoSapiens.is_evoluido()}') ⑤
    print(f'Neanderthal evoluído? {Neanderthal.is_evoluido()}') ⑤
    print(f'José evoluído? {jose.is_evoluido()}') ⑤
    print(f'Grokn evoluído? {grokn.is_evoluido()}') ⑤
```

① *Decorator* para métodos estáticos (`__builtins__`)

② *Decorator* para métodos de classe (`__builtins__`)

③ Método de classe que recebe pelo menos o parâmetro da classe que disparou o método, o que permite algum polimorfismo

④ Chamada de método estático (com diversas origens diferentes)

⑤ Chamada de método de classe (com diversas origens diferentes)

14.3. Propriedades

O uso de propriedades é um recurso bastante comum, que normalmente tem como objetivo proteger atributos da instância ou transformá-los na saída (leitura). Em algumas linguagens a sintaxe entre propriedades e atributos da instância é diferente, exigindo que a aplicação de propriedades precise ser feita o mais no início possível.

Em Python temos diversas abordagens para trabalhar com propriedades, mas via de regra a sugestão é não usá-las até que seja absolutamente necessária. Como na sua forma mais comum não há diferença de sintaxe entre um atributo ou propriedade, uma mudança tardia nesta questão normalmente não gera impactos.

Uma propriedade normalmente tem um *getter* e um *setter*, quando tem apenas um dos dois é *read-only* ou *write-only* respectivamente.

Exercício 100 - Propriedades através de métodos

evolucao_v3.py

```
class EvolucaoHumana(object):
    def __init__(self, nome):
        self.nome = nome
        self._idade = None ①

    def get_idade(self): ②
        return self._idade

    def set_idade(self, idade): ③
        if idade < 0: ④
            raise ValueError('Idade deve ser um número positivo!')
        self._idade = idade

if __name__ == '__main__':
    jose = HomoSapiens('José')
    jose.set_idade(40) ⑤
    print(f'Nome: {jose.nome} Idade: {jose.get_idade()}') ⑥
```

- ① Criação do atributo “privado” para ser usado pela propriedade idade
- ② *Getter*
- ③ *Setter*
- ④ Validação, apenas números positivos são aceitos como idade
- ⑤ Chamando o *setter*
- ⑥ Chamando o *getter*

Exercício 101 - Utilizando o *decorator* @property

evolucao_v4.py

```
class EvolucaoHumana(object):
    def __init__(self, nome):
        self.nome = nome
        self._idade = None

    @property ①
    def idade(self): ①
        return self._idade

    @idade.setter ②
    def idade(self, idade): ②
        if idade < 0:
            raise ValueError('Idade deve ser um número positivo!')
        self._idade = idade

if __name__ == '__main__':
    jose = HomoSapiens('José')
    jose.idade = 40 ③
    print(f'Nome: {jose.nome} Idade: {jose.idade}') ③
```

① *Decorator* property como uma forma mais simples e direta de utilizar um descritor de propriedade, neste caso o nome do método será o nome da propriedade

② A própria nova propriedade também possui o setter *decorator*, que permite definir que método será o *setter*, o nome deste método não importa, mas o melhor é manter o mesmo nome

③ O uso de uma propriedade assim não difere de um atributo comum, seja para leitura ou atribuição



Existem diversas outras maneiras de se utilizar propriedades em Python.
Mais informações em <https://docs.python.org/3/howto/descriptor.html>.

14.4. Classe abstrata

O suporte para classes abstratas não é nativa da linguagem, sendo adicionada a posteriori, tendo atualmente algumas implementações viáveis.

A primeira delas é para atender a necessidade de marcar métodos como abstratos (que precisam ser definidas nas classes descendentes), que é atingida apenas levantando uma exceção `NotImplementedError` em qualquer tentativa de chamada.

Exercício 102 - Método abstrato usando NotImplementedError

evolucao_v5.py

```
class EvolucaoHumana(object):
    @property
    def inteligente(self):
        raise NotImplementedError('Propriedade não implementada!') ①

class Neanderthal(EvolucaoHumana):
    especie = EvolucaoHumana.especies()[-2]

    @property
    def inteligente(self):
        return False ②

class HomoSapiens(EvolucaoHumana):
    especie = EvolucaoHumana.especies()[-1]

    @property
    def inteligente(self):
        return True ③

if __name__ == '__main__':
    anonimo = EvolucaoHumana('John Doe')
    try:
        print(anonimo.inteligente) ④
    except NotImplementedError:
        print('propriedade abstrata')

    jose = HomoSapiens('José')
    print(f'{jose.nome} da classe {jose.__class__.__name__}, inteligente: {jose.inteligente}') ④

    grogn = Neanderthal('Grogn')
    print(f'{grogn.nome} da classe {grogn.__class__.__name__}, inteligente: {grogn.inteligente}') ④
```

- ① Levantar exceção ao executar o método da classe abstrata
- ② Implementação da método abstrato nos descendentes
- ③ Tentativa de acessar método diretamente em uma classe abstrata
- ④ Acesso a propriedade reimplementada nos descendentes

Exercício 103 - Método abstrato usando o módulo abc: Abstract Base Class

evolucao_v6.py

```
from abc import ABCMeta, abstractproperty ①

class EvolucaoHumana(object, metaclass=ABCMeta): ②
    @abstractproperty ③
    def inteligente(self):
        pass

class Neanderthal(EvolucaoHumana):
    especie = EvolucaoHumana.especies()[-2]

    @property
    def inteligente(self):
        return False

class HomoSapiens(EvolucaoHumana):
    especie = EvolucaoHumana.especies()[-1]

    @property
    def inteligente(self):
        return True

if __name__ == '__main__':
    try:
        anonimo = EvolucaoHumana('John Doe') ④
        print(anonimo.inteligente)
    except TypeError:
        print('classe abstrata')

    jose = HomoSapiens('José')
    print(f'{jose.nome} da classe {jose.__class__.__name__}, inteligente: {jose.inteligente}')

    grogn = Neanderthal('Grogn')
    print(f'{grogn.nome} da classe {grogn.__class__.__name__}, inteligente: {grogn.inteligente}')
```

- ① Uso do módulo abc da biblioteca padrão para trabalhar com *abstract classes*
- ② Definição da metaclass da classe EvolucaoHumana, tornando-a uma classe abstrata
- ③ Definindo a propriedade como abstrata, o que exige uma implementação nos descendentes
- ④ Usando o módulo abc a própria inicialização de um objeto de uma classe abstrata já gera um TypeError

14.5. Herança Múltipla

Supor a herança é um item fundamental em qualquer linguagem que suporte orientação a objetos, porém o mais comum é a herança simples, poucas linguagens suportam herança múltipla.

Este recurso também não é muito popular, já que aumenta muito a complexidade do código por conta da resolução do polimorfismo, e aqui não é muito diferente, e por isso devemos evitá-lo.

Porém se necessário ele sempre estará por aqui.

Na sintaxe em vez de uma única classe base entre os parenteses, podemos incluir várias separadas por vírgulas. A ordem define a sequência de resolução, por isso é extremamente importante, a classe mais revelante deve ficar mais a direita da lista e os que estão a sua esquerda podem sobrescrever métodos e chamar os mais básicos (a direita) através do `super()`.

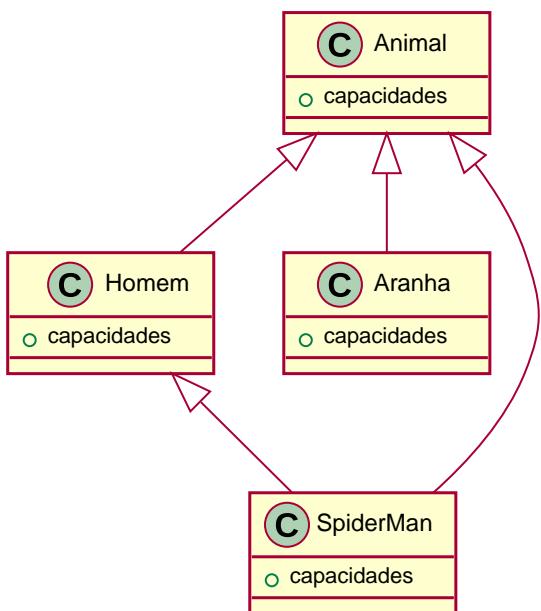


Figura 3. Diagrama para herança múltipla

Exercício 104 - Herança múltipla

multiple.py

```
#!/usr/bin/python3

class Animal(object):
    @property
    def capacidades(self): ①
        return ('dormir', 'comer', 'beber')

class Homem(Animal):
    @property
    def capacidades(self):
        return super().capacidades + ('amar', 'andar', 'correr') ②

class Aranha(Animal):
    @property
    def capacidades(self):
        return super().capacidades + ('fazer teia', 'andar pelas paredes') ②

class SpiderMan(Aranha, Homem):
    @property
    def capacidades(self):
        return super().capacidades + ('bater em bandidos', 'atirar teias entre prédios') ②

if __name__ == '__main__':
    peter = SpiderMan()
    print(f'Peter: {peter.capacidades}')

    john = Homem()
    print(f'John: {john.capacidades}')

    aranha = Aranha()
    print(f'Aranha: {aranha.capacidades}')
```

① Definição da propriedade capacidades

② Sobrescrita da propriedade capacidades, adicionando antes todas as capacidades herdadas (tanto na horizontal quanto vertical)

14.6. Mixins

É uma técnica de reuso de código, que inclui determinados comportamentos em uma classe, que pode ser aplicado via herança múltipla. Em Python diversos frameworks utilizam esta capacidade em suas API's.

Como prática os mixins devem herdar diretamente de `object`, evitando assim o aumento de complexidade.

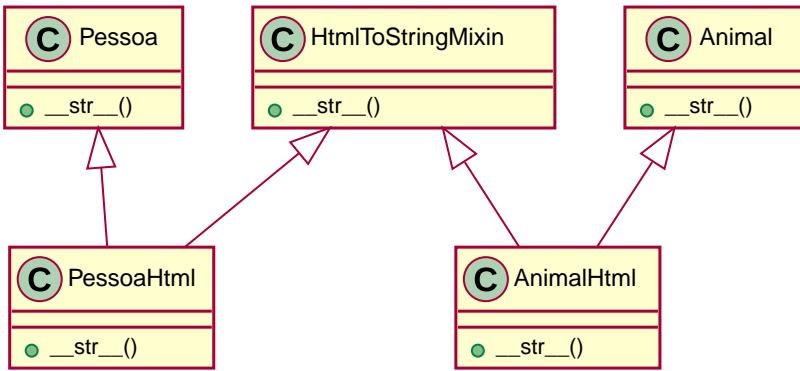


Figura 4. Diagrama do exercício de Mixins

Exercício 105 - Mixins

mixins.py

```
#!/usr/bin/python3

class HtmlToStringMixin(object):
    def __str__(self): ①
        """Conversão para HTML"""
        html = super().__str__() \
            .replace('(', '<strong>(') \
            .replace(')', ')</strong>')

        return f'<span>{html}</span>'

class Pessoa(object):
    def __init__(self, nome):
        self.nome = nome

    def __str__(self):
        return self.nome

class Animal(object):
    def __init__(self, nome, pet=True):
        self.nome = nome
        self.pet = pet

    def __str__(self):
        return self.nome + ' (pet)' if self.pet else ''

class PessoaHtml(HtmlToStringMixin, Pessoa): ②
    pass

class AnimalHtml(HtmlToStringMixin, Animal):
    pass

if __name__ == '__main__':
    leo = Pessoa('Leonardo Leitão')
    print(leo)

    juracy = PessoaHtml('Juracy Filho')
    print(juracy)

    toto = AnimalHtml('Totó')
    print(toto)
```

① Definição do `__str__` convertendo para HTML

② Criação da classe com uso do *mixin*

14.7. Protocolo Iterator

Durante todo o curso vimos várias formas de iteração, em Python temos um protocolo (algo similar a interfaces para *Duck Typing*) para objetos iteráveis, basta implementar o método `next` e levantar uma exceção `StopIteration` para finalizar (é plenamente aceitável *iterators* infinitos, que nunca levantam essa exceção).

Outro caso comum são objetos que podem ser convertidos em iteráveis, normalmente eles implementam o método `__iter__`, que é chamado pela função `iter`.

Várias classes da biblioteca padrão ou são iteráveis ou podem ser convertidos. Também já vimos formas simplificadas de criar *iterators*, como: [Generators](#) e [Generator Expression](#).

Exercício 106 - Iterator

iterator.py

```
#!/usr/bin/python3

class RGB(object):
    def __init__(self):
        self.cores = ['red', 'green', 'blue'][::-1]

    def __next__(self):
        try:
            return self.cores.pop() ①
        except IndexError: ②
            raise StopIteration() ③

if __name__ == '__main__':
    cores = RGB()
    print(next(cores)) ④
    print(next(cores))
    print(next(cores))
    try:
        print(next(cores)) ⑤
    except StopIteration:
        print('- acabou o iteration')
```

① Recupera o último elemento da lista, removendo-o

② Ao tentar recuperar um elemento de uma lista vazia é levantado um *IndexError*

③ Não tendo mais elementos, levanta `StopIteration`

④ Imprime o próximo elemento do iterator

⑤ `StopIteration` é levantado após a exaustão das opções

Habilidades adquiridas

Conseguimos agora evoluir nosso conhecimento no suporte a programação orientada a objetos em Python, podendo atingir resultados bastante profissionais com este paradigma. Tivemos os seguintes destaques:

- *Decorators*

`classmethod`

Decorator disponível no `__builtins__` para transformar um método em método de classe;

`staticmethod`

Decorator disponível no `__builtins__` para transformar um método em método estático;

`property`

Decorator disponível no `__builtins__` para transformar um método em uma propriedade;

`abstractproperty`

Decorator disponível no módulo `abc` para transformar uma propriedade em abstrata;

- Compreensão sobre a diferença entre membros de classe e de instância, e o lookup automático através da instância;
- Tipos de método;
- Criação e uso de propriedades;
- Classes e métodos abstratos, usando duas maneiras diferentes;
- Herança múltipla e *mixins*.

Desafio

Utilizando algum recurso deste capítulo registrar a quantidade de instâncias criadas de uma determinada classe.

Exemplo de chamada da nova classe

```
if __name__ == '__main__':
    lista = [SimpleClass(), SimpleClass()]
    print(SimpleClass.count) # Esperado 2
```



Existem técnicas mais adequadas para este fim.

Exemplo de solução disponível em [Desafio 10 - Contador de objetos](#)

Anexo A: Soluções

Área do Quadrado

Desafio 1 - Cálculo da área do círculo ou quadrado

area.py

```
#!/usr/bin/python3
import math
import sys

def circulo(raio):
    return math.pi * raio ** 2

def quadrado(lado):
    return lado ** 2

def help():
    print("""\
Sintaxe:
    area circulo <raio>
ou
    area quadrado <lado>""")

if __name__ == '__main__':
    if len(sys.argv) < 3:
        help()
        print('Nem todos os parâmetros foram informados')
        sys.exit(1)

    if sys.argv[1] not in ('circulo', 'quadrado'):
        help()
        print('O primeiro parâmetro deve ser circulo ou quadrado')
        sys.exit(2)

    if not sys.argv[2].isnumeric():
        help()
        print('O raio/lado deve ser um valor inteiro')
        sys.exit(2)

    if sys.argv[1] == 'circulo':
        raio = int(sys.argv[2])
        area = circulo(raio)
        print('Área do círculo', area)
    else:
        lado = int(sys.argv[2])
        area = quadrado(lado)
        print('Área do quadrado', area)
```

Fibonacci

Desafio 2 - Fibonacci

desafio_fibonacci.py

```
#!/usr/bin/python3

def is_fibonacci(numero):
    sequencia = [0, 1]

    while sequencia[-1] < numero:
        sequencia.append(sum(sequencia[-2:]))

    return numero in sequencia

if __name__ == '__main__':
    import sys

    numero = int(sys.argv[1])
    if is_fibonacci(numero):
        print(numero, 'faz parte da sequência de fibonacci!')
    else:
        print(numero, 'não faz parte da sequência de fibonacci!')
```

Manipulação de arquivos

Desafio 3 - Tratamento de CSV

io_desafio_1.py

```
#!/usr/bin/python3
import csv

def read(arquivo):
    with open(arquivo, encoding='latin1') as entrada:
        for cidade in csv.reader(entrada):
            print(f'{cidade[8]}: {cidade[3]}')


if __name__ == '__main__':
    import sys
    read(sys.argv[1])
```

Tabuada com *List Comprehension*

Desafio 4 - Tabuada

desafio_comprehension.py

```
#!/usr/bin/python3

print('\n'.join(f'{x} x {y} = {x*y}' for x in range(1, 10) for y in range(1, 10)))
```

Desafio 5 - MDC

desafio_mdc.py

```
#!/usr/bin/python3

def mdc(args):
    def calc(divisor):
        return divisor if sum(map(lambda x: x % divisor, args)) == 0 else calc(divisor - 1)
    return calc(min(args))

if __name__ == '__main__':
    print(mdc([21, 7])) # 7
    print(mdc([125, 40])) # 5
    print(mdc([9, 564, 66, 3])) # 3
    print(mdc([55, 22])) # 11
    print(mdc([15, 150])) # 15
    print(mdc([7, 9])) # 1
```

Gerador de HTML

Desafio 6 - Gerador de HTML

desafio_html.py

```
#!/usr/bin/python3

def tag(tag, *args, **kwargs):
    if 'css' in kwargs:
        kwargs['class'] = kwargs.pop('css')
    attrs = ''.join(f'{k}="{v}"' for k, v in kwargs.items())
    inner = ''.join(args)
    return f'{tag} {attrs}>{inner}</{tag}>'

if __name__ == '__main__':
    print(
        tag('p',
            tag('span', 'Curso de Python 3, por '),
            tag('strong', 'Juracy Filho', id='jf'),
            tag('span', ' e '),
            tag('strong', 'Leonardo Leitão', id='ll'),
            tag('span', '.'),
            css='alert')
    )
```

Palavras proibidas com set

Desafio 7 - Palavras proibidas com set

desafio_set.py

```
PALAVRAS_PROIBIDOS = {'futebol', 'religião', 'politica'}
```

```
textos = [
    'João gosta de futebol e politica',
    'A praia foi divertida',
]
```

```
for texto in textos:
    intersecao = PALAVRAS_PROIBIDOS.intersection(set(texto.lower().split()))
    if intersecao:
        print('Texto possui palavras proibidas:', intersecao)
    else:
        print('Texto autorizado:', texto)
```

Criação de um pacote

Desafio 8 - Pacote app

app/__init__.py

```
app/negocio/__init__.py
```

```
def check_exists(nome):
    return False
```

app/negocio/backend.py

```
def add_nome(nome):
    pass
```

app/utils/__init__.py

app/utils/generators.py

```
from random import choice

def nome_proprio():
    return choice(['Juracy', 'Leonardo', 'Pedro', 'João'])
```

Controle de vendas de uma loja

Desafio 9 - Controle de vendas

loja/__init__.py

```
from .cliente import Cliente
from .vendedor import Vendedor
from .compra import Compra

# A classe Pessoa não foi exposta propositalmente (pois não é necessária)
__all__ = ['Cliente', 'Vendedor', 'Compra']
```

loja/compra.py

```
class Compra(object):
    def __init__(self, vendedor, data, valor):
        self.vendedor = vendedor
        self.data = data
        self.valor = valor
```

loja/pessoa.py

```
MaiorIdade = 18

class Pessoa(object):
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def __str__(self):
        if not self.idade:
            return self.nome

        return f'{self.nome} ({self.idade} anos)'

    def isAdult(self):
        return (self.idade or 0) > MaiorIdade
```

loja/cliente.py

```
from .pessoa import Pessoa
from functools import reduce

class Cliente(Pessoa):
    def __init__(self, nome, idade):
        super().__init__(nome, idade)
        self.compras = []

    def registrar_compra(self, compra):
        self.compras.append(compra)

    def get_data_ultima_compra(self):
        return None if not self.compras else sorted(self.compras, key=lambda compra: compra.data)[-1].data

    def total_compras(self):
        return reduce(lambda c1, c2: c1 + c2, (compra.valor for compra in self.compras))
```

loja/vendedor.py

```
from .pessoa import Pessoa

class Vendedor(Pessoa):
    def __init__(self, nome, idade, salario):
        super().__init__(nome, idade)
        self.salario = salario
```

desafio_loja.py

```
from datetime import datetime
from loja import Cliente, Vendedor, Compra

def main():
    juracy = Cliente('Juracy Filho', 44)
    leo = Vendedor('Leonardo Leitão', 36, 1000)
    compra1 = Compra(leo, datetime.now(), 512)
    compra2 = Compra(leo, datetime(2018, 6, 4), 256)
    juracy.registrar_compra(compra1)
    juracy.registrar_compra(compra2)

    print(f'Cliente: {juracy}', '(adulto)' if juracy.isAdult() else '')
    print(f'Vendedor: {leo}')
    print(f'Total: {juracy.total_compras()} em {len(juracy.compras)} compras')
    print(f'Última compra: {juracy.get_data_ultima_compra()}')

if __name__ == '__main__':
    main()
```

Contador de objetos

Desafio 10 - Contador de objetos

contador_objetos.py

```
#!/usr/bin/python3

class SimpleClass(object):
    count = 0

    def __init__(self):
        self.inc()

    @classmethod
    def inc(cls):
        cls.count += 1

if __name__ == '__main__':
    lista = [SimpleClass(), SimpleClass()]
    print(SimpleClass.count) # Esperado 2
```

Anexo B: Exemplos avançados

Fibonacci

Exemplo Avançado 1 - Fibonacci recursivo decrescente sem *memoize*

ex-fibonacci_recursive_decrescente.py

```
#!/usr/bin/python3

def fib(n):
    return n if n in (0, 1) else fib(n-1) + fib(n-2)

if __name__ == '__main__':
    # Vigésimo (começando de zero)
    # Sem memoize a função fib é executada 13529 vezes
    print(fib(20 - 1))
```

Fibonacci com *memoize*

Exemplo Avançado 2 - Fibonacci recursivo com decorators, *memoize* e classes

ex-fibonacci_recursive_memoize.py

```
#!/usr/bin/python3
from functools import wraps

class MemoizeStopCondition(object):
    """Decorator memoize genérico com cache inicial"""

    def __init__(self, stop_conditions=None):
        self.cache = stop_conditions or {}

    def __call__(self, fn):
        @wraps(fn)
        def decorated(*args):
            key = tuple(args)
            if key not in self.cache:
                self.cache[key] = fn(*args)
            return self.cache.get(key)
        return decorated

    @MemoizeStopCondition({
        (0,): 0,
        (1,): 1
    })
def fib(n):
    """Função recursiva com condição de parada no cache"""
    return fib(n-1) + fib(n-2)

if __name__ == '__main__':
    # Com memoize a função fib é executada apenas 18 vezes
    for i in range(20):
        print(fib(i))
```

Tratamento de CSV com *download*

Exemplo Avançado 3 - Tratamento de CSV com download

io_desafio_2.py

```
#!/usr/bin/python3

import csv
from urllib import request

def read(url):
    with request.urlopen(url) as entrada:
        print('Baixando o CSV...')
        dados = entrada.read().decode('latin1')
        print('Download completo!')

    for cidade in csv.reader(dados.splitlines()):
        print(f'{cidade[8]}: {cidade[3]}')


if __name__ == '__main__':
    read(r'http://www.geoservicos.ibge.gov.br/geoserver/wms?service=WFS&version=1.0.0&request=GetFeature&typeName=CGEO:Re
deUrbanaSintese_Regic2007&outputFormat=CSV')
```

MDC

Exemplo Avançado 4 - MDC Funcional

mdc_funcional.py

```
#!/usr/bin/python3

def mdc(args):
    # next -> tuple()[0] - com a vantagem de processar apenas o primeiro elemento (iterator)
    return next(
        map(
            lambda y: y[0],
            filter(
                lambda x: x[1],
                map(
                    lambda divisor: (divisor, sum(map(lambda x: x % divisor, args)) == 0),
                    range(min(args), 0, -1)
                )
            )
        )
    )

if __name__ == '__main__':
    print(mdc([21, 7])) # 7
    print(mdc([125, 40])) # 5
    print(mdc([9, 564, 66, 3])) # 3
    print(mdc([55, 22])) # 11
    print(mdc([15, 150])) # 15
    print(mdc([7, 9])) # 1
```

Várias soluções para fatorial

Exemplo Avançado 5 - Fatorial

fatorial.py

```
#!/usr/bin/python3
from math import factorial
from functools import reduce
from random import choice
from operator import mul

def loops(n):
    if n < 0:
        return None

    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

def funcional(n):
    if n < 1:
        return None if n < 0 else 1

    return reduce(lambda x, y: x * y, range(1, n + 1))

def funcional2(n):
    if n < 1:
        return None if n < 0 else 1

    return reduce(mul, range(1, n + 1))

def recursivo(n):
    if n < 1:
        return None if n < 0 else 1

    return n * recursivo(n - 1)

if __name__ == '__main__':
    functions = [factorial, loops, funcional, funcional2, recursivo]

    for i in range(20):
        func = choice(functions)
        print('{0:2d} {1:15s} {2:18d}'.format(i, func.__name__, func(i)))
```

Solução recursiva para a Torre de Hanoi

Exemplo Avançado 6 - Torre de Hanoi

hanoi.py

```
#!/usr/bin/python3

def hanoi(n, A, B, C):
    if n > 0: # Condição de parada
        hanoi(n-1, A, C, B)
        print(f'Mova o disco {n} de {A} para {B}')
        hanoi(n-1, C, B, A)

if __name__ == '__main__':
    hanoi(4, 'um', 'dois', 'três')
```

Anexo C: Listas auxiliares

Listas de tabelas

- [Tipos básicos de dados](#)
- [Operadores](#)
- [Literais para inteiros em outras bases numéricas](#)

Listas de figuras e diagramas

- [Snake trap](#)
- [Diagrama de classes do desafio de OO](#)
- [Diagrama para herança múltipla](#)
- [Diagrama do exercício de *Mixins*](#)

Anexo D: Listagem de Códigos

Exercícios

- [Exercício 1 - Alô Mundo](#)
- [Exercício 2 - Atribuição](#)
- [Exercício 3 - Função type\(\)](#)
- [Exercício 4 - TypeError](#)
- [Exercício 5 - Conversão de tipos](#)
- [Exercício 6 - Coerção de tipos](#)
- [Exercício 7 - Números](#)
- [Exercício 10 - Listas](#)
- [Exercício 11 - Indexação das Listas](#)
- [Exercício 12 - Fatiamento de Listas](#)
- [Exercício 13 - Tuplas](#)
- [Exercício 14 - Dicionários](#)
- [Exercício 15 - Atualização nos Dicionários](#)
- [Exercício 16 - Área do círculo - versão 1](#)
- [Exercício 17 - Área do círculo - versão 2](#)
- [Exercício 18 - Área do círculo - versão 3](#)
- [Exercício 19 - Área do círculo - versão 4](#)
- [Exercício 20 - Área do círculo - versão 5](#)
- [Exercício 21 - Área do círculo - versão 5 \(correção\)](#)
- [Exercício 22 - Área do círculo - versão 6](#)
- [Exercício 23 - Área do círculo - versão 7](#)
- [Exercício 24 - Área do círculo - versão 8](#)
- [Exercício 25 - Área do círculo - versão 9](#)
- [Exercício 26 - Área do círculo - versão 10](#)
- [Exercício 27 - Área do círculo - versão 11](#)
- [Exercício 28 - Área do círculo - versão 12](#)
- [Exercício 29 - Área do círculo - versão 13](#)
- [Exercício 30 - Área do círculo - versão 14](#)
- [Exercício 31 - Fibonacci - While infinito](#)
- [Exercício 32 - Fibonacci - While condicional](#)
- [Exercício 33 - Fibonacci - Uso do packing](#)

- Exercício 34 - Fibonacci - Iterando uma lista
- Exercício 35 - Fibonacci - sum
- Exercício 36 - Fibonacci - break
- Exercício 37 - Fibonacci - range
- Exercício 38 - Fibonacci recursivo
- Exercício 39 - Fibonacci recursivo com operador ternário
- Exercício 40 - Leitura Básica de Arquivo
- Exercício 41 - Leitura Básica de Arquivo (*stream*)
- Exercício 42 - Leitura Básica de Arquivo (*stream*) — Fix
- Exercício 43 - Mais robustez com try..finally
- Exercício 44 - Leitura de Arquivo com with
- Exercício 45 - Gravação de Arquivo
- Exercício 46 - Leitura de Arquivo com o módulo csv
- Exercício 47 - Dobros
- Exercício 48 - Dobros dos pares
- Exercício 49 - Generators
- Exercício 50 - Generators com for
- Exercício 51 - Dict Comprehension
- Exercício 52 - Funções de primeira classe
- Exercício 53 - Funções de alta ordem
- Exercício 54 - Funções com escopos aninhados (*closure*)
- Exercício 55 - Totalização de compras (*lambda*)
- Exercício 56 - Totalização de compras sem o uso de lambda
- Exercício 57 - Cálculo de fatorial usando recursividade
- Exercício 58 - Listar todos os meses do ano com 31 dias
- Exercício 59 - Listar todos os meses do ano com 31 dias (funcional em uma linha)
- Exercício 60 - Listar todos os meses do ano com 31 dias (imperativo)
- Exercício 61 - Diversas funções úteis trabalham com objetos imutáveis
- Exercício 64 - Implementação do *generator* map
- Exercício 65 - Implementação do map com *generator expression*
- Exercício 66 - Parâmetros opcionais (com valores *default*)
- Exercício 67 - Argumentos nomeados
- Exercício 68 - *Unpacking* de argumentos
- Exercício 69 - Combinando *unpacking* e parâmetros opcionais
- Exercício 70 - *Unpacking* de argumentos nomeados

- Exercício 71 - Objetos chamáveis
- Exercício 72 - Problemas com argumentos mutáveis ☠
- Exercício 73 - Problemas com argumentos mutáveis (solução)
- Exercício 74 - Decorator `log`
- Exercício 75 - Conhecendo o `elif`
- Exercício 76 - Simulando um `switch`
- Exercício 77 - `Switch` com valores únicos
- Exercício 78 - `Switch` baseado em faixa de valores
- Exercício 79 - Conhecendo a fundo o `while`
- Exercício 80 - Conhecendo a fundo o `for`
- Exercício 81 - Uso de variável de controle extra no `for`
- Exercício 82 - Uso do `else` no `for`
- Exercício 83 - Conhecendo a fundo o `try`
- Exercício 84 - Execução de uma função em outro `package`
- Exercício 85 - Momento de execução do código
- Exercício 86 - Uso de módulos com mesmo nome
- Exercício 87 - Importação direta das funções no `namespace` atual
- Exercício 88 - Uso de um pacote como `façade`
- Exercício 89 - Consumidor do pacote do desafio
- Exercício 90 - Classe `Task`
- Exercício 91 - Classe `Project`
- Exercício 92 - Método `__iter__`
- Exercício 93 - Implementação do vencimento (`datetime` e `timedelta`)
- Exercício 94 - Herança
- Exercício 95 - Métodos “privados” e simulação de “overload”
- Exercício 96 - Sobrecarga de operador
- Exercício 97 - *Snake trap* - Tratamento de exceções
- Exercício 98 - Membros de classe × membros da instância
- Exercício 99 - Tipos de métodos
- Exercício 100 - Propriedades através de métodos
- Exercício 101 - Utilizando o `decorator @property`
- Exercício 102 - Método abstrato usando `NotImplementedError`
- Exercício 103 - Método abstrato usando o módulo `abc: Abstract Base Class`
- Exercício 104 - Herança múltipla
- Exercício 105 - *Mixins*

- Exercício 106 - *Iterator*

Soluções de desafios

- Desafio 1 - Cálculo da área do círculo ou quadrado
- Desafio 2 - Fibonacci
- Desafio 3 - Tratamento de CSV
- Desafio 4 - Tabuada
- Desafio 5 - MDC
- Desafio 6 - Gerador de HTML
- Desafio 7 - Palavras proibidas com set
- Desafio 8 - Pacote app
- Desafio 9 - Controle de vendas
- Desafio 10 - Contador de objetos

Exemplos avançados

- Exemplo Avançado 1 - Fibonacci recursivo decrescente sem *memoize*
- Exemplo Avançado 2 - Fibonacci recursivo com decorators, *memoize* e classes
- Exemplo Avançado 3 - Tratamento de CSV com *download*
- Exemplo Avançado 4 - MDC Funcional
- Exemplo Avançado 5 - Fatorial
- Exemplo Avançado 6 - Torre de Hanoi

Glossário

Zen of Python

Criado por Tim Peters, representa os alicerces fundamentais da linguagem.

Ofidioglossia

É o idioma de serpentes (bem como outras criaturas à base de serpente mágicas, como o farosutil) e aqueles que podem conversar com eles. Uma pessoa que pode falar a língua das cobras é conhecido como um Ofidioglota. É uma habilidade muito rara, e normalmente é hereditária. Quase todos os ofidioglotos conhecidos são descendentes de Salazar Sonserina, com Harry Potter sendo uma notável exceção.— <http://pt-br.harrypotter.wikia.com/wiki/Ofidioglossia>

Módulo

XXX

Lisp

É uma família de linguagens de programação concebida por **John McCarthy** em 1958. Num célebre artigo, ele mostra que é possível usar exclusivamente funções matemáticas como estruturas de dados elementares (o que é possível a partir do momento em que há um mecanismo formal para manipular funções: o Cálculo Lambda de **Alonzo Church**).— [Wikipédia](#)

Garbage Colector

TODO

Parâmetro

TODO

Argumento

TODO

Importante conhecer



Glossário da documentação oficial: <https://docs.python.org/3/glossary.html>