



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 🧘	3
2. Python <i>everywhere</i> 🌎	4
2.1. Implementações	4
2.2. Python 2? No thanks! 💀	4
2.3. Instalação	5
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>	11
4.4. Conversão de tipos e coerções	14
4.5. Números	17
4.6. Strings	18
4.7. Listas	20
4.8. Tuplas	23
4.9. Dicionários	24
4.10. Conjuntos	27
4.11. Interpolações de <i>Strings</i>	30
5. Ninho de cobras	31
5.1. Primeiro módulo	31
5.2. <i>Encoding</i>	31
5.3. <i>Shebang</i>	32
5.4. Baterias inclusas	32
5.5. Entrada de dados	33
5.6. Nome do módulo	34
5.7. E se	34
5.8. Quebrando nosso código em funções	35
5.9. Funções podem retornar valores	36
5.10. Passando argumentos pela linha de comando	37
5.11. Verificação dos argumentos	37
5.12. Melhorando um pouco o nosso <i>help</i>	38
5.13. Dando retorno ao sistema operacional	39

5.14. Validando argumentos	40
6. Instruções	43
6.1. Conhecendo o <code>while</code>	43
6.2. Execute enquanto	43
6.3. <i>Packing</i>	44
6.4. Conhecendo o <code>for</code>	45
6.5. Totalizando uma lista	45
6.6. Forçando a quebra de um laço	46
6.7. Gerando uma lista de números	46
6.8. Recursão	47
6.9. Operador ternário ... <code>if ... else ...</code>	49
7. Manipulação de arquivos	51
7.1. Leitura	51
7.2. Gravação	53
7.3. Arquivos separados por vírgula	54
8. Comprehension	56
9. Programação funcional	58
9.1. Capacidades implementadas	58
9.2. <i>First Class Functions</i> - Funções de primeira classe	58
9.3. <i>High Order Functions</i> - Funções de alta ordem	59
9.4. <i>Closure</i> - Funções com escopos aninhados	60
9.5. <i>Anonymous Functions</i> - Funções anônimas (<code>lambda</code>)	61
9.6. Recursion - Recursividade	63
9.7. <i>Immutability</i> - Imutabilidade	64
9.8. <i>Lazy Evaluation</i> - Avaliação preguiçosa	68
9.9. Nem tudo são flores... ☺	72
10. Às funções e além!	76
10.1. Tipos de parâmetros	76
10.2. Parâmetros nomeados ou opcionais	77
10.3. Argumentos nomeados	77
10.4. <i>Unpacking</i> de argumentos	78
10.5. Combinando <i>unpacking</i> e parâmetros opcionais	79
10.6. <i>Unpacking</i> de argumentos nomeados	79
10.7. Objetos chamáveis	81
10.8. Problemas com argumentos mutáveis	81
10.9. Decorators	83
11. Dominando as instruções Python	86
11.1. E se... senão se...	86
11.2. <i>Switch? Case?</i> Não, obrigado!	87
11.3. Laços condicionais <i>like a boss</i>	88
11.4. Iterações <i>like a boss</i>	89

11.5. Tratamento de exceções <i>like a boss</i>	91
12. Packages	93
13. Programação orientada a objetos	96
13.1. Classe Task	96
13.2. Classe Project	98
13.3. Método <code>__iter__()</code>	99
13.4. Implementação do vencimento	100
13.5. Herança	101
13.6. Métodos “privados”	102
13.7. Sobrecarga de operador	103
13.8. <i>Snake trap</i>	104
14. Orientada a objetos - Avançado	108
14.1. Membros de classe × membros da instância	108
14.2. Métodos em profundidade	109
14.3. Propriedades	111
14.4. Classe abstrata	112
14.5. Herança Múltipla	115
14.6. <i>Mixins</i>	116
14.7. Protocolo <i>Iterator</i>	119
15. Gerenciamento de pacotes	121
15.1. <code>pip</code>	121
16. Isolamento de Ambientes	127
16.1. <code>venv</code>	127
17. Banco de dados	131
17.1. PEP 249 — Python Database API Specification v2.0	131
17.2. Preparação do ambiente	131
17.3. Configuração do acesso ao banco de dados	132
17.4. Criação do nosso banco de dados	132
17.5. Uso do nosso banco de dados: agenda	134
17.6. Manipulação de dados	138
17.7. Seleção de dados	139
17.8. Associação	145
17.9. SQLite	149
Anexo A: Soluções	154
Área do Quadrado	154
Fibonacci	155
Manipulação de arquivos	155
Tabuada com <i>List Comprehension</i>	156
MDC	156
Gerador de HTML	156
Palavras proibidas com <code>set</code>	157

Criação de um pacote	158
Controle de vendas de uma loja	158
Contador de objetos	160
Lista de tarefas persistente	161
Anexo B: Exemplos avançados	162
Fibonacci	162
Fibonacci com <i>memoize</i>	162
Tratamento de CSV com <i>download</i>	163
MDC	164
Várias soluções para fatorial	165
Solução recursiva para a Torre de Hanoi	166
Anexo C: Listas auxiliares	167
Lista de tabelas	167
Lista de figuras e diagramas	167
Anexo D: Listagem de Códigos	168
Exercícios	168
Soluções de desafios	172
Exemplos avançados	172
Glossário	173

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** ou o **jupyter** e seus *notebooks*.



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

- ❶ 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
- ❷ 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

coercão_automatica.py

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

- ❶ A partir da versão 3, a divisão mesmo entre números inteiros sempre retorna um float (*ponto flutuante*)
- ❷ O operador // realiza a divisão de números e sempre retorna um int, truncando se necessário
- ❸ Em operações numéricas, objetos do tipo bool, retornam 1 para True e 0 para False
- ❹ 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 na seção [Instruções](#).

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 apostrófós 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] ❶  
'Juracy'  
>>> 're' in nome ❷  
False  
>>> 'ra' in texto  
True  
>>> len(nome) ❸  
12  
>>> nome.lower() ❹  
'juracy filho'  
>>> nome.upper() ❺  
'JURACY FILHO'  
>>> nome.split() ❻  
['Juracy', 'Filho']
```

- ❶ *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](#)
- ❷ O operador `in` avalia se a primeira *string* está contida na segunda, retornando um booleano
- ❸ 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
- ❹ O método `lower()` retorna uma nova *string* com todos os caracteres em minúsculo
- ❺ O método `upper()` retorna uma nova *string* com todos os caracteres em maiúsculo
- ❻ 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](#).



Na seção [Interpolações de Strings](#) veremos várias técnicas de formatação de strings.

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) ❶
[..., 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> len(lista) ❷
0
>>> lista.append(1) ❸
>>> lista.append(5)
>>> lista
[1, 5]
>>> len(lista)
2
>>> nova_lista = lista + ['Juracy', 'Leonardo', 3.1415] ❹
>>> nova_lista
[1, 5, 'Juracy', 'Leonardo', 3.1415]
>>> nova_lista.insert(0, 'Zero') ❺
>>> nova_lista
['Zero', 1, 5, 'Juracy', 'Leonardo', 3.1415]
>>> nova_lista.remove(5) ❻
>>> nova_lista
['Zero', 1, 'Juracy', 'Leonardo', 3.1415]
```

- ❶ Lista dos membros (*métodos, constantes, ...*) disponíveis para o tipo `list`, uma dica é utilizar a função `help(list)` que detalhará melhor cada um deles
- ❷ 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
- ❸ O método `append()` inclui um novo elemento na lista
- ❹ O uso do operador `+` com duas listas irá retornar uma nova lista juntando o conteúdo da primeira com a segunda (*sem alterar nenhuma delas*)
- ❺ O método `insert()` inclui um novo elemento em uma posição específica da lista, lembrando que a primeira posição começa em 0
- ❻ O método `remove()` remove 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') ❶
2
>>> lista.index(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 42 is not in list ❷
>>> lista[2]
'Juracy'
>>> 1 in lista ❸
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 ❹
>>> lista[-1] ❺
3.1415
>>> lista[-5]
1
```

- ❶ O método `index` retorna o índice de um elemento indicado
- ❷ Executar o método `index` com um valor não pertencente a lista retornará um `ValueError`
- ❸ O operador `in` retorna se um objeto está contido na lista
- ❹ Ao tentar acessar um índice inexistente na lista, normalmente maior que o número de elementos, o Python levanta uma exceção do tipo `IndexError`
- ❺ 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 pega 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 41 - 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

4.10. Conjuntos

Os conjuntos são similares as listas, porém possui diferenças marcantes:

- Elementos únicos
- Não indexado
- Não ordenado
- Todos os elementos precisam ser imutáveis (não aceita dicionários e listas por exemplo)
- Possui diversos métodos baseado em lógica matemática de conjuntos (interseção, união, pertence, etc)

Exercício 16 - Conjuntos

conjuntos.py

```
>>> conjunto = set() ❶
>>> type(conjunto)
<class 'set'>
>>> type({})
>>> dir(conjunto) ❷
[..., 'add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update',
 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update',
 'union', 'update']
>>> conjunto = {2, 4, 5, 4} ❸
>>> len(conjunto)
3
>>> conjunto[0] ❹
TypeError: 'set' object does not support indexing
>>> a = set('codddd3r') ❺
>>> print(a)
{'r', '3', 'c', 'd', 'o'} ❻
>>> print('3' in a, 4 not in a) ❼
True True
>>> {1, 2, 3} == {3, 2, 1, 3} ❽
True
```

- ❶ Conjunto vazio, a partir da chamada da classe `set`
- ❷ Não há maneira literal de declarar um `set`, `{}` é um dicionário vazio
- ❸ Lista dos membros (*métodos, constantes, ...*) disponíveis para o tipo `set`, uma dica é utilizar a função `help(set)` que detalhará melhor cada um deles
- ❹ Sintaxe literal para conjuntos, lembrando que elementos duplicados são ignorados
- ❺ Conjuntos não aceitam indexação
- ❻ O `set` aceita qualquer sequência como entrada, inclusive uma *string*
- ❼ A ordem não é garantida
- ❽ Operador `in` funciona como esperado
- ❽ Numa comparação de igualdade, a ordem dos elementos não importa e as duplicações são ignoradas

Exercício 17 - Conjunto (operações)

conjunto-operacoes.py

```
>>> c1 = {1, 2}
>>> c2 = {2, 3}
>>> c1.union(c2) ①
{1, 2, 3}
>>> c1.intersection(c2) ②
{2}
>>> c1.update(c2) ③
>>> c1
{1, 2, 3}
>>> c2 <= c1 ④
True
>>> c1 >= c1 ⑤
True
>>> c1 - c2
>>> c1 -= {2} ⑥
```

- ① Método `union`, retorna um novo set com a união dos elementos de `c1` e `c2`
- ② Método `intersection`, retorna um novo set com a interseção dos elementos de `c1` e `c2`
- ③ Altera o conjunto `c1` incluindo todos os elementos de `c2`
- ④ Indica se `c2` contém todos os elementos de `c1` (*superset*)
- ⑤ Indica se todos os elementos de `c1` estão contidos em `c2` (*subset*)
- ⑥ Reatribui a diferenças entre os conjuntos ao `c1`

Tabela 4. Equivalência entre operadores × métodos no set

Operador	Método	Descrição
Operadores binários		
	<code>union</code>	União de conjuntos
&	<code>intersection</code>	Interseção entre os conjuntos
-	<code>difference</code>	Diferença entre os conjuntos (elementos do primeiro que não estão no segundo)
^	<code>symmetric_difference</code>	Diferença simétrica (o inverso da interseção, apenas os elementos que estão somente no primeiro ou somente no segundo conjunto)
Operadores relacionais		
<code><=</code>	<code>issubset</code>	Verdadeiro se o primeiro conjunto estiver contido no segundo
<code>>=</code>	<code>issuperset</code>	Verdadeiro se o primeiro conjunto contiver o segundo

Operador	Método	Descrição
Operadores de atribuição		
=	update	Atribui a união dos conjuntos (antes e depois do operador) a variável antes do operador
&=	intersection_update	Atribui a interseção dos conjuntos a variável antes do operador
-=	difference_update	Atribui a diferença dos conjuntos a variável antes do operador
^=	symmetric_difference_update	Atribui a diferença simétrica dos conjuntos a variável antes do operador

4.11. Interpolações de Strings

Os interpolações ou formatações de texto são extremamente úteis, podem ser simuladas utilizando concatenação simples de *strings*, mas os resultados são muito inferiores que as que veremos abaixo.

Exercício 18 - Interpolações

interpolacoes.py

```
from string import Template ①

nome, idade = 'Ana', 30
print('Nome: %s Idade: %d' % (nome, idade)) # mais antiga ②
print('Nome: {0} Idade: {1}'.format(nome, idade)) # python < 3.6 ③
print(f'Nome: {nome} Idade: {idade}') # python >= 3.6 ④

s = Template('Nome: $nome Idade: $idade') ⑤
print(s.substitute(nome=nome, idade=idade)) ⑥
```

① Importação do classe `Template` do módulo `string`

② Interpolação com o operador módulo `%` (legado)

③ Interpolação com a função `str.format()`, método preferencial até o Python 3.5, tendo ainda vários usos como na internacionalização e sempre que a avaliação precise ser adiada.

④ O novo padrão de interpolação, chamado também de *f-string*, só está disponível a partir do Python 3.6 e sempre processa a *string* imediatamente (*eager evaluation*)

⑤ Formato de interpolação muito usado para tratar com segurança dados entrados pelo usuário

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.

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 uma IDE, neste curso utilizamos o Microsoft Visual Source Code (**vscode** para os íntimos).

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

5.1. Primeiro módulo

Exercício 19 - 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 263 para uma explicação completa.

Vejamos a solução aplicando o *encoding*.

Exercício 20 - 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 21 - 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, simplificando 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 22 - 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 23 - 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 23 - 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 24 - 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 25 - 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 25 - 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 26 - 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 26 - Área do círculo - versão 7](#) ajustamos o nosso módulo para ser executado apenas através da chamada direta, porém ficamos sem nenhuma funcionalidade ao importá-lo. Agora podemos dividir nosso código em funções, permitindo que as funções possa ser executadas por outros módulos.

Exercício 27 - 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 27 - 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 28 - 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 29 - 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 e propriedades relevantes: `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 é 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 30 - Área do círculo - versão 11](#) deixaria o usuário bastante confuso, no próprio exercício veremos um tratamento melhor para isso.

Exercício 30 - 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("""\
É 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 31 - 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(): ❶
    print("""\
É necessário informar o raio do círculo.

Sintaxe: area_circulo <raio>""")

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

❶ Nova função `help` para impressão da ajuda.

❷ 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 32 - 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 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 33 - 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.

Recursos externos

Modules 

<https://docs.python.org/3/tutorial/modules.html>

PEP 263 — *Defining Python Source Code Encodings*

<https://www.python.org/dev/peps/pep-0263>

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 34 - 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 35 - 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 36 - 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 37 - 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 38 - 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 39 - 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 40 - 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 41 - 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 <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 42 - 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:])))\n\nif __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

Python possui um extenso suporte para manipulação de arquivos. Apesar de existir um módulo específico para isso, o `io`, o `__builtins__` possui um atalho para a principal função: `open()` que instanciará a classe correta conforme os argumentos passados. E por isso muitas vezes não precisamos de uma importação explícita deste módulo.

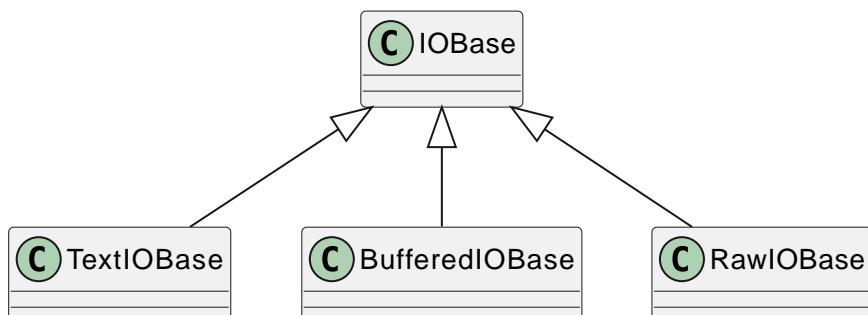


Figura 1. Diagrama das classes de manipulação de arquivo

7.1. Leitura

Exercício 43 - Leitura Básica de Arquivo

`io_v1.py`

```
#!/usr/bin/python3

arquivo = open('pessoas.csv') ①
dados = arquivo.read() ②
arquivo.close() ③

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

- ① A função `open()` do `__builtins__` (que é um atalho para o `io.open()`), abre o arquivo passado como argumento e retorna uma instância do `io.TextIOWrapper`
- ② O método `read()` lê todo o conteúdo do arquivo
- ③ O método `close()` fecha o arquivo

Exercício 44 - Leitura Básica de Arquivo (stream)

io_v2.py

```
#!/usr/bin/python3

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

- ① O objeto que representa o arquivo pode ser lido linha a linha, como neste `for` (conceito de *iterator* que veremos mais adiante no curso), extremamente útil em arquivos grandes, não apenas pelo uso otimizado de memória, como pelo inicio mais imediato da execução do laço (sem precisar ler todo o arquivo para memória antes de começar)

Exercício 45 - 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(','))) ①
arquivo.close()
```

- ① Lendo como um *iterator* traz também o delimitador de linha (*carriage return* e/ou *line feed*), o uso do `strip()` remove estes caracteres.

Exercício 46 - Mais robustez com try...finally

io_v4.py

```
#!/usr/bin/python3

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

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

- ① Uma vez que o arquivo é aberto, o mesmo precisa ser fechado para liberar recursos do sistema operacional. O `try...finally` executa o bloco `try` e depois o bloco `finally`, mesmo que haja erro no primeiro bloco, ou seja o bloco `finally` é de execução obrigatória
② Fecha o arquivo, liberando os seus recursos

Exercício 47 - 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 A instrução `with...as` cria um bloco especial de contexto, associado a um determinado objeto. Ao entrar no bloco o método `__enter__()` do objeto é executado, e ao encerrar (ou mesmo em caso de erro) o método `__exit__()`, simulando um `try...finally`. Isso torna essa instrução excelente para manipulação limitada de um objetos em determinados contextos

② A propriedade `closed` indica se o arquivo está fechado

7.2. Gravação

Exercício 48 - 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 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. Outra opção é abri-lo em modo binário (`'rb'` e `'wb'`), essencial para arquivos como imagens, vídeos, áudios, etc

7.3. Arquivos separados por vírgula

Exercício 49 - 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 O módulo `csv` da biblioteca padrão para tratamento de arquivos CSV (*Comma-separated values* ou valores separados por vírgulas)
- ② O método `reader()` retorna um *iterator* (como o objeto de arquivo também o faz), e pode ser iterado pelo `for`

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

Recursos externos 🔎

Desafio 🏆

Extrair 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.



Por segurança temos uma cópia deste arquivo no nosso servidor, que pode ser baixado em <http://files.cod3r.com.br/curso-python/desafio-ibge.csv>. Isso é importante em vários casos como indisponibilidade ou até reestruturação do site do IBGE.

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 50 - Dobros

comprehension_v1.py

```
#!/usr/bin/python3

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

Exercício 51 - 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 52 - 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 53 - 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 54 - 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 55 - 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'O {func.__name__} de {numero} é {func(numero)}') ③ ④
```

- ① Operador de multiplicação em listas, gerando uma nova lista de 10 elementos (repetindo a lista original 5×)
- ② A função `zip()` do `_builtins_`, permite combinar dois ou mais iteráveis em um, gerando um *iterator* que retorna tuplas, a iteração termina assim que algum dos objetos originais se exaurir;
- ③ Toda função é um objeto do tipo/classe `function`, e entre as propriedades destes objetos temos `__name__` que possui o nome da função
- ④ Todo objeto do tipo `function` é *callable*, ou seja, possui o método `__call__()` que também pode ser mais brevemente chamado adicionando parenteses após o objeto, assim: `func()`

9.3. High Order Functions - Funções de alta ordem

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

Exercício 56 - 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 Aproveitando funções `dobro` e `quadrado`, importando-as do módulo `first_class_functions`
- ② Recebendo no parâmetro `funcao`, um argumento do tipo `function`, o que representa exatamente uma **função de alta ordem**
- ③ Chamando a função recebida, através do parâmetro `funcao`
- ④ 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 Collector](#) de liberá-las.

Exercício 57 - 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 O *closure* em si, função aninhada com acesso a variável `times` do escopo da função `multiplier()`
- ② A execução acessa a variável `times`, que retém o seu valor
- ③ Retorna a função aninhada, que pode ser chamada externamente e utilizar o valor original da `times`. Enquanto este valor retornado não for coletado (pelo GC), todas as outras variáveis deste escopo também não serão
- ④ Conhecendo um pouco das funções criadas: `dobra()` e `triplo()`, e as suas representações mostram que as mesmas foram criadas internamente na função `multiplier()`
- ⑤ Ao chamar a função `triplo()`, o valor passado será multiplicado por 3, que é o valor original do `times` para este **closure**
- ⑥ Ao chamar a função `dobra()`, o valor passado será multiplicado por 2, que é o valor original do `times` para este outro **closure**

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 58 - Totalização de compras (`lambda`)

`lambda_functions.py`

```
#!/usr/bin/python3

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

❶ Criação de uma tupla de dicionários, com as compras a serem totalizadas

❷ A função `map()` retorna um *iterator*, e aqui convertemos em uma tupla para sua total realização... Basicamente fazendo uma *eager evaluation*, isso é necessário por que queremos percorre-la duas vezes no final, gerar uma lista (ou poderíamos imprimir a tupla mesmo) e fazer um somatório

❸ Usamos a função `map()`, que recebe dois argumentos, uma `function` e um ou mais `iterables`. Para cada iteração, é executada a função passando o(s) elemento(s) da iteração passados como parâmetro

❹ Em vez de passar uma função já definida, podemos passar uma função anônima inline. Neste caso o produto da quantidade e preço de cada compra.

❺ A tupla de totais é totalizada, usando a função `sum()` que itera em cada elemento acumulando seu total

Exercício 59 - Totalização de compras sem o uso de lambda

lambda_functions_alternative.py

```
#!/usr/bin/python3

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

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

- ❶ Definição da função `calc_preco_total()` gerando o produto da quantidade e preço
- ❷ Substituição da `lambda` pela identificador da função `calc_preco_total()`

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 60 - Cálculo de fatorial usando recursividade

fatorial_recurssivo.py

```
#!/usr/bin/python3

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

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

① Chamada recursiva a função factorial()

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 41 - Fibonacci recursivo](#)
- [Exercício 42 - 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 61 - 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 O módulo `locale` da biblioteca padrão, possui diversas funcionalidades relativas a internacionalização (i18n) e localização. Neste caso estamos usando o `setlocale` e a constante `LC_ALL`

② O módulo `calendar` possui diversas funções auxiliares para impressão de calendário, incluindo aí traduções e *helpers* como os nomes do meses (`month_name`), e a quantidade de dias de cada mês (`mdays`), conforme o `locale`.

③ A função `functools.reduce` (que no **Python 2** ficava no `__builtins__`), permite a iteração de um *iterable* qualquer de forma acumulativa, pegando o resultado da última iteração e passando como parâmetro para a próxima, em conjunto com o elemento atual sendo iterado.

④ Configura o `locale` atual no processo em execução para `pt_BR` (usando `UTF-8` como *encoding*)

⑤ A função `reduce()` recebe três argumentos:

1. Função que será usada para o processamento (redução em um único valor);
2. Objeto que será iterado (*iterable*);
3. Valor inicial, que será usado como o acumulado anterior para o primeiro item.

⑥ Função anônima usada pelo `reduce()` que concatena o texto acumulado anterior e adiciona uma nova linha com o nome do mês

⑦ A função `map()` aqui é usada para mapear um faixa de 1 a 12 no seu nome de mês equivalente

⑧ Função anônima usada pelo `map()` para converter o número do mês em seu nome

⑨ A função `filter()` está selecionando para a função `map()` apenas aqueles meses que tem 31

dias

- ⑩ Função anônima usada pelo `filter()` que retorna `True` apenas para os meses com 31 dias
- ⑪ Faixa iterável de 1 a 12 (tamanho da coleção `month_name`, que é 13, pois o mês 0 está em branco)
- ⑫ Valor inicial usado pelo `reduce()`, basicamente servirá como texto inicial antes do primeiro mês ser adicionado ao resultado

Exercício 62 - 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 63 - Listar todos os meses do ano com 31 dias (imperativo)

imperativo.py

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

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

# 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 64 - 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 O módulo `operator` da biblioteca padrão, possui funções de baixo nível equivalentes aos mais diversos operadores, neste caso usaremos a função `add()`, que equivale a soma.
- ② Tupla com os valores que são processados, usamos aqui um `tuple` para reforçar a ideia de imutabilidade
- ③ A função `sorted()` retorna uma lista ordenada com os valores encontrados no *iterable* de entrada. Diferentemente do `list.sort()`, ela não alterar o objeto que originou a ordenação
- ④ A função `min()` retorna o menor valor de um *iterable*
- ⑤ A função `max()` retorna o maior valor de um *iterable*
- ⑥ Totaliza todos os valores de um *iterable*, seja usando a função `sum()` ou o `reduce()`
- ⑦ Podemos simular o mesmo resultado da função `sum()` através da função `reduce()`, que aplica acumuladamente uma outra função em um *iterable*, retornando um valor final, neste caso aplicamos a soma: `operator.add()`
- ⑧ Temos aqui uma classe, que basicamente é usada como uma função chamada `reversed`, que retorna um *iterator*, que irá percorrer *on demand* um *iterable* passado como argumento em ordem inversa
- ⑨ Aqui temos um *eager evaluation* de uma reversão de valores de um *iterable*, através da transformação em uma tupla



Todas as funções e classes acima funcionam sem alterar o objeto original, retornando um novo objeto. Cumprindo assim o princípio da imutabilidade

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 65 - 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 A instrução `yield` retorna um valor e suspende a execução da função atual mantendo seu estado até que uma próxima iteração solicite um novo valor, quando a sua execução continuará a partir da próxima linha. Caso encontre um novo `yield`, o ciclo continua, caso contrário, chegue ao término da função (equivalente a `return None`) ou execute uma instrução `return` explícita, ocorrerá um `StopIteration` encerrando o *iterator*
- ② Observe que o retorno da função é um objeto da classe `generator`
- ③ Laço infinito (teoricamente)
- ④ Consumindo o próximo valor do *iterator* (neste caso um `generator`), através da função `next()`



Como não há tratamento, esta rotina se encerra com um `StopIteration` quando não houverem mais valores para consumir

Exercício 66 - 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 Uso da instrução `for` para consumir um *iterator*, não precisando chamar explicitamente a função `next()`
- ② Consumindo um *iterator* através da instrução `for`, o próprio Python se encarrega de parar o laço quando não mais houverem valores a consumir

Exercício 67 - Implementação do *generator map*

map.py

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

def mapear(function, lista): ①
    """Implementação simplificada do map"""
    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 Implementação simplificada da função `map()`, utilizando a instrução `for` para consumir um *iterable* e executando a função recebida como parâmetro
- ② Utilizamos a instrução `yield` para retornar *on demand* o processamento dos dados originais um a um
- ③ Podemos assim substituir a função original `map()` pela nossa `mapear()`



Apesar de ser possível essa substituição, a função original `map()` é extremamente otimizada, testada e documentada, não justificando sua troca sem um excelente motivo

9.8.2. Generator Expression

Exercício 68 - Implementação do `map` com *generator expression*

`map_generator_expression.py`

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

def mapear(function, lista):
    """Implementação simplificada do map"""
    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 Podemos usar uma sintaxe similar ao *list comprehension*, chamada *generator expression*, para substituir as instruções `for` e `yield`

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 ou mais iteráveis em um, gerando um *iterator*, que termina de iterar assim que algum dos objetos originais se exaurir;

`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.*

Recursos externos

Functional Programming HOWTO 

<https://docs.python.org/3/howto/functional.html>

Functional Programming Modules 

<https://docs.python.org/3/library/functional.html>

Higher-order functions and operations on callable objects 

<https://docs.python.org/3/library/functools.html>

Operator — Standard operators as functions 

<https://docs.python.org/3/library/operator.html>

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 69 - 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 70 - 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} class="{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 Novo parâmetro nomeado: `inline`, que indicará se será usado o `` ou `<div>`
- ② Definição da `tag` principal conforme parâmetro `inline`
- ③ É possível passar uma argumento posicional para o `inline`, sendo o terceiro
- ④ Podemos nomear nossos argumentos para definir (independente da ordem) determinados parâmetros na função

10.4. Unpacking de argumentos

Exercício 71 - 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} class="{classe}">{texto}</{tag}>'

def build_list(*itens): ①
    lista = ''.join(f'- {item}
' for item in itens)
    return f'

{lista}
'

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_list('Sábado', 'Domingo'), classe='info') ②
```

- ① Nova função que monta uma lista em HTML (baseada em `` e ``) com todos os argumentos recebidos (no parâmetro `itens`, no caso uma `tuple`)
- ② O conteúdo do bloco será uma lista com os itens: Sábado e Domingo

10.5. Combinando *unpacking* e parâmetros opcionais

Exercício 72 - 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'- {item}
' for item in itens)
    return f'

{lista}
'

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')) ④
```

- ① A função `build_block()` agora mapeia todos os argumentos posicionais (a partir do segundo) no parâmetro `args`
- ② O conteúdo agora pode ser uma função (na realidade um objeto “chamável”, ou seja, que implemente o método `__call__()`), e será chamado com o parâmetro `args` para gerar o conteúdo real
- ③ Devido à mudança na assinatura da função `build_block()`, os argumentos "info" e `True` foram mapeados para o argumento `args` e como o conteúdo não é uma função foram ignorados
- ④ Agora é possível passar para o conteúdo, a próprio função `build_list()` e todos os argumentos mapeados em `args` irão compor uma lista em HTML

10.6. *Unpacking* de argumentos nomeados

Exercício 73 - 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}>' ⑥

def build_list(*itens, **kwargs): ④
    lista = ''.join(f'<li {filteredAttrs(kwargs, li_attributes)}>{item}</li>' for item in itens) ⑥
    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 Tuplas para listar os atributos (argumentos nomeados) que serão aceitos
- ② Nova função `filteredAttrs()` que recebe um dict (`kwargs`) e uma lista de atributos a serem filtrados/aceitos a partir do dicionário
- ③ Inclusão do parâmetro `kwargs` que recebe todos os argumentos nomeados ainda não mapeados nos outros parâmetros nomeados como um dicionário
- ④ A função que responde pelo conteúdo, agora também receberá os argumentos nomeados, no caso do `build_list()`, como um dicionário
- ⑤ Como os atributos da `tag` agora podem ser povoados, a classe CSS também usará o mesmo método
- ⑥ Chamada da nova função `filteredAttrs()` para filtrar a partir dos argumentos passados quais são atributos válidos (conforme a lista de atributos aceitos)
- ⑦ Os argumentos nomeados serão distribuídos (conforme cada `tag`), neste caso a `<div>` com `class` e `accesskey` e a `` com `type`

10.7. Objetos chamáveis

Exercício 74 - Objetos chamáveis

callable_object.py

```
#!/usr/bin/python3

class ClosureClass(object): ❶
    """Calcula uma potência específica"""\n\n    ❷\n\n    def __init__(self, potencia): ❸\n        self.potencia = potencia\n\n    def __call__(self, valor): ❹\n        return valor ** self.potencia\n\n\nif __name__ == '__main__':\n    quadrado = ClosureClass(2) ❺\n    cubo = ClosureClass(3)\n\n    if callable(quadrado):\n        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)}')
```

- ❶ Veremos classes com mais detalhes em [Programação orientada a objetos](#)
- ❷ Documentação da classe: `__doc__`
- ❸ Constructor da classe, que recebe a potência
- ❹ Implementação do método especial `__call__()` indica que o objeto poderá ser chamado como uma função
- ❺ 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 75 - 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 76 - 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 77 - 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.

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 78 - 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 79 - 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 80 - 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 81 - 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 82 - 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 83 - 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 84 - Uso de variável de controle extra no for

for_sem_else.py

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

textos = [
    'João gosta de futebol e política',
    '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 85 - 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 84 - Uso de variável de controle extra no for](#) e [Exercício 85 - 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 86 - 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.

Recursos externos 🔎

Compound statements 💬

https://docs.python.org/3/reference/compound_stmts.html

12. Packages

Exercício 87 - 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 88 - 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 89 - 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 90 - 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 91 - 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, partícula `as`.

Desafio

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

Exercício 92 - 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 93 - Classe Task

todo_v1.py

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

class Task(object): ① ②
    def __init__(self, descricao): ③ ④
        self.descricao = descricao ⑤
        self.feito = False
        self.criacao = datetime.now()

    def done(self):
        self.feito = True

    def __str__(self): ⑥
        return f'{self.descricao}' + (' (feito)' if self.feito else '')

def main():
    casa = []
    casa.append(Task('Passar roupa')) ⑦
    casa.append(Task('Lavar prato'))
    [task.done() for task in casa if task.descricao == 'Lavar prato'] ⑧
    for task in casa:
        print(f'- {task}')

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

- ① A instrução `class` é usada para definir uma classe, por convenção as classes definidas além da biblioteca padrão devem usar *camel case*, neste caso `Task`
- ② Podemos definir de que outra classe iremos herdar, a classe mais básica é `object`, porém a partir do **Python 3** não precisamos definir a classe base caso a mesma seja `object`. Esta linha poderia ser: `class Task:`
- ③ Uma função dentro de uma classe, é chamada de método, e terá pelo menos um parâmetro (o primeiro) que receberá o objeto instanciado que disparou a execução, isso é automático, o usuário da classe só passará argumentos a partir do segundo parâmetro. Por convenção este primeiro parâmetro é nomeado como `self`
- ④ O método `__init__()` é disparado logo após a instanciação do objeto, seria algo equivalente a um construtor
- ⑤ Atribuições de membros em `self`, inicializa atributos da instância
- ⑥ O método `__str__()` é chamado ao tentar converter o objeto em *string*, ou tentar imprimi-lo por exemplo (que faz implicitamente essa conversão)
- ⑦ A forma de instanciar um objeto de uma classe é chamar a classe de maneira análoga a uma função, passando argumentos se necessário
- ⑧ O método `task.done()` altera o atributo de instância `feito` para `True`

13.2. Classe Project

Exercício 94 - Classe Project

todo_v2.py

```
class Project: ①
    def __init__(self, nome):
        self.nome = nome
        self.tasks = [] ②

    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: ③
        print(f'- {task}')

    print(mercado)
    for task in mercado.tasks: ③
        print(f'- {task}')
```

- ① Como dito anteriormente, é possível declarar uma classe sem informar seu ancestral, e neste caso é assumido `object`
- ② Na classe `Project` teremos uma lista de tarefas que pertencem ao projeto
- ③ É possível acessar o atributo `tasks` e iterar sobre eles

13.3. Método `__iter__()`

Exercício 95 - Método `__iter__()`

todo_v3.py

```
class Project:
    def __iter__(self):
        return iter(self.tasks) ①

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: ③
        print(f'- {task}')

    print(mercado)
    for task in mercado: ③
        print(f'- {task}')
```

- ① A presença do método `__iter__()` transformar o objeto é um *iterable*, ou seja um iterável, o retorno dele deve ser um *iterator* o que significa que suportará o método `__next__()`, tanto um método quanto o outro podem ser acessados pelas funções `iter()` e `next()` respectivamente
- ② Neste caso o *iterator* retornado pelo objeto será o mesmo da lista `tasks`, que poderia ser chamado pelo seu método mágico: `self.tasks.__iter__()`, porém o uso da função `iter()` é mais aconselhado
- ③ Agora podemos iterar diretamente sobre os objetos da classe `Project`, sem precisar conhecer sua estrutura interna e usando uma forma considerada mais pythônica

Duck typing

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

— James Whitcomb Riley

13.4. Implementação do vencimento

Exercício 96 - Implementação do vencimento (datetime e timedelta)

todo_v4.py

```
from datetime import datetime, timedelta ①

class Task:
    def __init__(self, descricao, vencimento=None): ②
        self.descricao = descricao
        self.feito = False
        self.criacao = datetime.now()
        self.vencimento = vencimento ②

    def __str__(self):
        decorators = [] ③
        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:
    def add(self, descricao, vencimento=None): ④
        self.tasks.append(Task(descricao, vencimento)) ④

def main():
    casa = Project('Casa')
    casa.add('Passar roupa')
    casa.add('Lavar prato')
    casa.add('Arrumar guarda-roupa', datetime.now() + timedelta(days=3)) ⑤
    casa.add('Pintar', datetime.now() - timedelta(days=7)) ⑥
```

① Nova função `datetime.timedelta`, que é responsável por calcular diferenças entre datas

② O constructor da classe `Task` agora suporta opcionalmente o vencimento da tarefa

③ A conversão para `string` agora inclui diversos decoradores, além de **feito**, temos **vencido** e **vence em**

④ O método `Project.add` também suporta opcionalmente a data de vencimento da tarefa

⑤ Esta tarefa vence em 3 dias

⑥ Tarefa vencida a uma semana

13.5. Herança

Exercício 97 - Herança

todo_v5.py

```
class TaskRecurring(Task): ①
    def __init__(self, descricao, vencimento, dias=7): ②
        super().__init__(descricao, vencimento)
        self.dias = dias

    def done(self):
        super().done()
        return TaskRecurring(self.descricao, datetime.now() + timedelta(days=self.dias), self.dias) ③

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)) ④

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

    print('*** Tarefa recorrente ***')
    casa.tasks.append(casa.find('Trocá lençóis').done()) ⑤
    for task in casa:
        print(f'- {task}')
```

- ① Nova classe `TaskRecurring`, herdando da classe `Task` e adicionando um suporte para recorrência
- ② O constructor dessa nova classe exige o vencimento e possui um novo parâmetro opcional para quantidade de dias para o próximo vencimento, por *default* 7 dias
- ③ Ao concluir um tarefa recorrente, uma nova tarefa recorrente é criada e retornada com o novo vencimento (baseado na quantidade de dias configurada no atributo `dias`)
- ④ Incluímos manualmente uma nova tarefa recorrente no projeto `casa`
- ⑤ Concluímos a tarefa recorrente e adicionamos a nova no projeto

13.6. Métodos “privados”

Exercício 98 - Métodos “privados” e simulação de “overload”

todo_v6.py

```
class Project:
    def __add_task(self, task, **kwargs): ①
        self.tasks.append(task)

    def __add_new_task(self, descricao, **kwargs): ②
        self.tasks.append(Task(descricao, kwargs.get('vencimento', None)))

    def add(self, task, vencimento=None, **kwargs): ③
        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)) ④

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

        print('*** Tarefa recorrente ***')
        casa.add(casa.find('Trocá lençóis').done()) ④
        for task in casa:
            print(f'- {task}')
```

- ① Novo método “privado” `Project.__add_task()`, em **Python** não existe membros de classe privados e totalmente inacessíveis externamente, há sim uma convenção de começar estes membros com *underline*. Este método adiciona na lista de tarefas, um objeto do tipo `Task`
- ② O método “privado” `Project.__add_new_task()` que cria uma nova `Task` baseado nos argumentos passados e adiciona-o na lista
- ③ O método `Project.add()` agora funciona como *hub*, simulando um “*overload*”, que dependendo dos argumentos passados chama o `Project.__add_task()` ou `Project.__add_new_task()`
- ④ Agora podemos usar o método `Project.add()` para adicionar as tarefas recorrentes, e não sendo necessário conhecer detalhes da implementação da classe `Project`

13.7. Sobrecarga de operador

Exercício 99 - 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 ①

    def done(self):
        super().done()
        new_task = TaskRecurring(self.descricao, datetime.now() + timedelta(days=self.dias), self.dias)
        if self.parent: ②
            self.parent += new_task
        return new_task


class Project:
    def __iadd__(self, task): ③
        task.parent = self ④
        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('Trocá lençóis', datetime.now(), 7) ⑤

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

    print('*** Tarefa recorrente ***')
    casa.find('Trocá lençóis').done() ⑥
    for task in casa:
        print(f'- {task}')
```

- ① A classe TaskRecurring agora tem um novo atributo parent para registrar o projeto que o adicionou em sua lista
- ② O método TaskRecurring.done() agora adiciona automaticamente a nova tarefa no projeto, caso o parent esteja setado
- ③ O método __iadd__() é disparado quando a atribuição com soma += é utilizada, deixando a classe mais pythônica
- ④ Seta o parent da Task adicionada
- ⑤ Substituição do uso do método add() pelo operador +=
- ⑥ Não é mais necessário chamar explicitamente o método add() ao chamar o TaskRecurring.done()

13.8. *Snake trap*

Exceções também são classes!



Figura 2. *Snake trap*

Exercício 100 - Snake trap - Tratamento de exceções

todo_v8.py

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

class Project:
    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('Trocá 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 Uma exceção é apenas uma classe herdada de `Exception`, aqui criamos o nossa para indicar que uma tarefa não foi encontrada
- ② A instrução `pass` indica que não há bloco de código específico
- ③ Já sabíamos que o `Project.find()` poderia levantar uma exceção `IndexError`, e aqui começamos o tratamento de erro com o `try...except...as`
- ④ No `except` podemos especificar que classe de exceções queremos capturar, neste caso `IndexError` e qualquer uma das suas especializações (heranças)
- ⑤ A instrução `raise` permite lançarmos uma exceção para o **Python**, que neste caso é a nossa própria exceção com a mesma mensagem da exceção original. Apesar de estarmos apenas substituindo que exceção precisará ser capturada, o `raise` pode ser executado nos mais diversos contextos, por exemplo, poderia ter um `if` e uma vez não atendida a condição, lançaríamos a exceção.
- ⑥ Agora podemos capturar nossa exceção `TaskNotFound`, quando for solicitado uma tarefa inexistente

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__()`

Suporte 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).

Recursos externos 🔎

Special Methods 🔎

<https://docs.python.org/3/reference/datamodel.html#specialnames>

Private Variables 🔎

<https://docs.python.org/3/tutorial/classes.html#private-variables>

Desafio 🏆

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

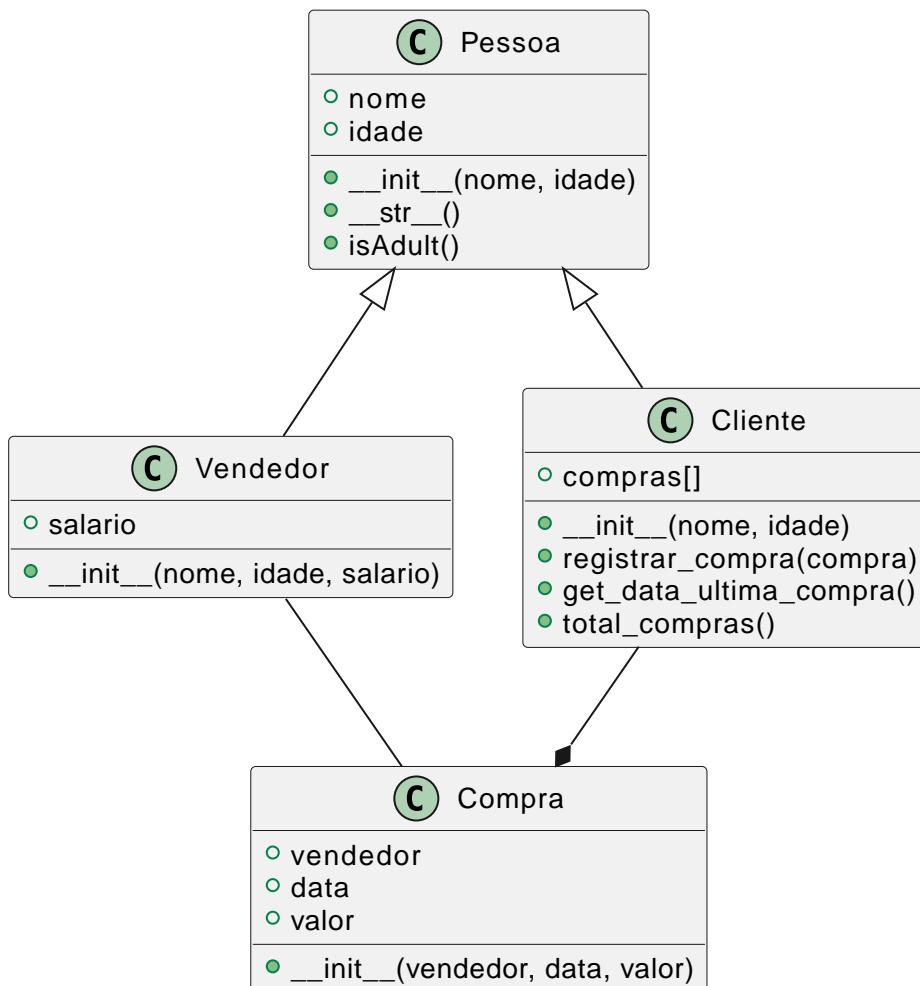


Figura 3. Diagrama de classes

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 101 - 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 possui-la, 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 102 - 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 103 - 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 104 - 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 105 - 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}') ⑤

    grogn = Neanderthal('Grogn')
    print(f'{grogn.nome} da classe {grogn.__class__.__name__}, inteligente: {grogn.inteligente}') ⑥
```

- 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 106 - Método abstrato usando o módulo abc: Abstract Base Class

evolucao_v6.py

```
from abc import ABCMeta, abstractmethod ①

class EvolucaoHumana(object, metaclass=ABCMeta): ②
    @property
    @abstractmethod 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: ④
        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 o 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 relevante deve ficar mais a direita da lista e os que estão a sua esquerda podem sobrepor métodos e chamar os mais básicos (a direita) através do `super()`.

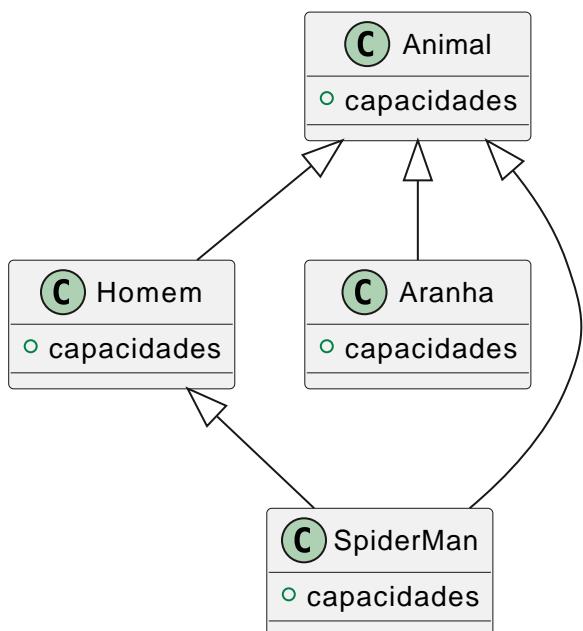


Figura 4. Diagrama para herança múltipla

Exercício 107 - 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.

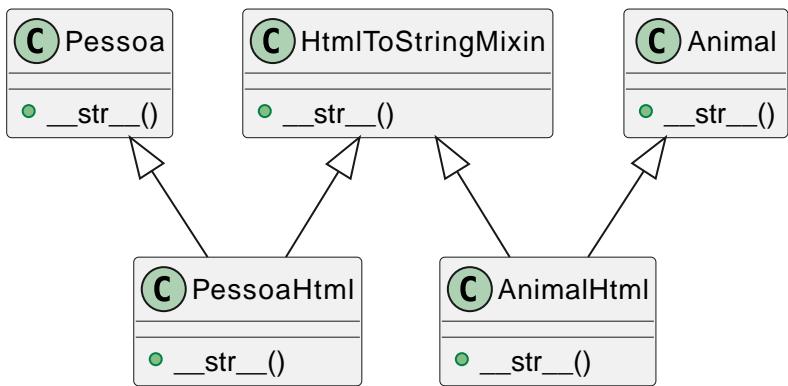


Figura 5. Diagrama do exercício de Mixins

Exercício 108 - 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 109 - *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](#)

15. Gerenciamento de pacotes

Python tem uma enorme comunidade ativa, e uma infinidade de *packages* com as mais diversas funcionalidades, que independente de estarem cobertas (ou não) pela biblioteca padrão, podem trazer maneiras mais interessantes, otimizadas ou apenas diferentes de se atingir um objetivo.

Existe um repositório público oficial da comunidade, o PyPI—*Python Package Index* em <https://pypi.org>. Atualmente existem mais de 190.000 *packages* ou projetos.

Talvez por conta da grande flexibilidade no sistema de importação de pacotes e sua fácil manipulação, por muito tempo o Python não teve uma forma padronizada de gerenciamento de pacotes.

Isso mudou a partir do Python 3.4, em que o **pip** (*Package Installer for Python*) passou a ser parte integrante da instalação e se tornou o programa preferido para gerir pacotes. Porém mesmo antes disso era possível instalá-lo ou usar outras alternativas como o **easy_install** por exemplo.

15.1. pip

Então vamos nos focar no **pip**, por fazer parte da biblioteca padrão.

15.1.1. Conferindo a versão

```
$ pip --version
```

Se por algum motivo o **pip** não seja encontrado (por exemplo não está no PATH), uma outra maneira de acessá-lo seria: `python -m pip --version`



O parâmetro `-m` permite rodar um módulo como *script*, ou seja o módulo terá o `__name__` como `__main__`. Este módulo será procurado na *path* de bibliotecas do Python.

A partir de 2018, o ano começou a fazer parte do número da versão, sendo assim essa versão 19.0.3 foi lançada em 2019.

15.1.2. Ajuda

O **pip** tem uma ajuda vasta. Que podemos ser consultada através do parâmetro `--help`.

Ajuda geral

```
$ pip --help
```

Aqui nos temos os principais comandos:

`install`

Instalar pacotes

uninstall

Desinstalar pacotes

list

Listar pacotes instalados

freeze

Listar pacotes instalados em formato para reinstalação futura

search

Procurar pacotes no PyPI

Podemos também consultar as opções de um comando específico:

Ajuda do comando de instalação

```
$ pip install --help
```

Aqui podemos destacar as opções: `--user`, `--upgrade` e `-r` (ou `--requirement`).

15.1.3. Listar pacotes instalados

Vamos começar listando os pacotes instalados atualmente no sistema.

```
$ pip list
```

Provavelmente você encontrará pacotes que você não conhece, simplesmente por que vieram junto com a sua instalação do Python, ou foram instalados como dependências de outros programas no sistema operacional, pois no momento não há isolamento dos pacotes (veremos mais adiante em [Isolamento de Ambientes](#)).

Diretório padrão dos pacotes externos

```
$ python -c "import site; print(site.getsitepackages()[0])"
```

O parâmetro `-c` permite enviar uma *string* para ser executada pelo Python, e sim, o Python suporta ponto e vírgula (`;`) para separar comandos na mesma linha. O que não é uma boa prática nos seus módulos, mas bastante útil nesses casos.

O módulo `site` faz a gestão dos pacotes de terceiros, e sabe aonde eles serão instalados, que podemos consultar pelo método `site.getsitepackages()`.

Rodar diretamente o módulo `site` como *script* traz diversas informações sobre o ambiente de pacotes do Python.

```
$ python -m site
```

No Linux é bem possível que o resultado seja algo similar a `/usr/lib/python3.7/site-packages`. E

aqui encontramos um primeiro problema neste formato, por `default` não temos isolamento, simplesmente todo e qualquer pacote será instalado aqui. E possivelmente para gravar neste diretório seja necessário uma conta de administrador.

Vamos experimentar instalar um pacote.

```
$ pip install Django
```

Ele vai começar a baixar o pacote (e suas dependências) e ao tentar gravar os novos pacotes, talvez falhe por falta de permissão:

```
Could not install packages due to an EnvironmentError: [Errno 13] Permissão negada
```

Existem várias maneiras de resolver isso, é possível instalar no *home* do usuário atual com `--user`).

```
$ pip install --user Django
```

Agora que ele instalou com sucesso, vamos entender o que ocorreu.

```
$ python -c 'import django; print(django)'
```

Aqui podemos ver o diretório aonde o pacote foi instalado, neste caso em um diretório oculto: `~/.local/lib/python3.7/site-packages`, no Windows seria `%APPDATA%/Python/Python37/site-packages`, conforme a PEP 370.



`%APPDATA%` é uma variável de ambiente no Windows que aponta para um diretório aonde gravar configurações e dados das aplicações.

```
$ python -c 'import sys; print(sys.path)' ①
```

- ① `sys.path` contém a sequência de caminhos usados na resolução dos *imports*, ou seja, toda vez que o Python precisa encontrar um módulo ou pacote ele percorre essa lista testando a existência do mesmo, levantando uma exceção caso não encontre em nenhum. Essa lista pode ser alterada de diversas formas, inclusive através da simples manipulação da mesma, como um `sys.path.insert`.

Aqui vemos que este diretório de pacotes no *home* vem antes do **site-packages** do sistema, portanto qualquer pacote no *home* terá preferência na ordem de busca de pacotes. Inclusive podendo ter o mesmo pacote em uma versão diferente (seja mais nova ou mais velha).

15.1.4. Desinstalação

O comando responsável por desinstalação é o `uninstall`.

```
$ pip uninstall Django  
$ python -m django  
  
/usr/bin/python: No module named django
```

Já não é mais possível importa-lo.

15.1.5. Congelar pacotes — `requirements.txt`

Além do `list` que lista as bibliotecas instaladas temos o `freeze`, que gera essa lista num formato replicável.

```
$ pip freeze
```

Neste formato ele especifica o nome do *package* e sua versão exata, o que permitirá replicar este ambiente futuramente.

```
$ pip freeze > requirements.txt
```

Temos a convenção de chamar este manifesto com os *packages* de `requirements.txt`, o que seria equivalente a chave *dependencies* do `package.json` no **npm**.

Agora podemos por exemplo remontar este ambiente:

```
$ pip install -r requirements.txt
```



Poderíamos trocar o `-r` por `--requirement`.

Neste caso todas as versões estavam idênticas as instaladas e não foi necessário instalar nada, vamos aplicar uma pequena mudança adicionando o `django`.

```
$ echo 'Django==2.2.4' >> requirements.txt  
$ pip install -r requirements.txt
```

E como esperado não foi possível por falta de permissão, vamos concluir no nosso usuário.

```
$ pip install --user -r requirements.txt
```

15.1.6. Pacotes desatualizados

Podemos listar quais são os pacotes desatualizados.

```
$ pip list --outdated
```

E podemos solicitar a atualização do pip.

```
$ pip install --user --upgrade pip
```

O pacote foi instalado, porém ainda não conseguimos acessa-lo diretamente.

```
$ pip --version
```

Continua com a versão anterior, por que o *shell* guardou no *cache* o caminho do anterior, porém:

```
$ python -m pip --version
```

Já consegue pegar o pacote novo na *home* e utiliza-lo. Podemos resolver rapidamente isso saindo e voltando ao *shell*.

Habilidades adquiridas

- Biblioteca padrão (*baterias incluídas*):

site

Módulo para gerir pacotes de terceiros;

site.getsitepackages()

Retorna o *path* atual do *site-packages*;

sys.path

Lista de *strings* com os caminhos usados na busca por pacotes e módulos;

- Pacotes de terceiros:

pip

Módulo para gerenciamento de pacotes;

- Parâmetros do interpretador Python:

-m

Rodar um módulo como *script*;

-c

Rodar um *script* Python passado como *string*;

- Casos de uso do pip:

```
install  
    Instala um pacote;  
  
install --user  
    Instala um pacote no home do usuário;  
  
install --upgrade  
    Atualiza um pacote;  
  
install -r  
    Instala pacotes a partir de um manifesto gerado pelo freeze;  
  
uninstall  
    Desinstala pacote;  
  
list  
    Lista pacotes instalados;  
  
list --outdated  
    Lista pacotes desatualizados;  
  
freeze  
    Gera manifesto com os pacotes instalados;
```

Recursos externos

Installing Python Modules 

<https://docs.python.org/3/installing/index.html>

PEP 370 — Per user site-packages directory

<https://www.python.org/dev/peps/pep-0370>

16. Isolamento de Ambientes

Conforme já estudamos, o (**pip**) não implementa isolamento de ambientes, apenas o gerenciamento de pacotes em si.

O isolamento de ambientes é muito útil em diversas situações como:

- Projetos com pacotes em versões diferentes das atuais (muitas vezes congeladas em uma versão em que foi homologado)
- Falta de clareza nos exatos requisitos de um projeto, já que todas os pacotes do sistema são listadas junto com as do projeto
- Projetos em diferentes versões do Python

Ao longo do tempo, diversas soluções foram criadas para ajudar a contornar os problemas acima, uma delas, o **Virtualenv**, serviu de base para o **venv** (na realidade um *subset* da primeira) que foi adotado pela biblioteca padrão do Python na versão 3.3.

Existem outras soluções além dessas duas como o **Conda**, **Pipenv**, ...

Vamos adotar aqui o **venv** por fazer parte da biblioteca padrão.



Algumas soluções permitem uma liberdade maior de seleção da versão do Python, infelizmente o **venv** só poderá usar versões instaladas no seu sistema e acima da 3.3.

16.1. venv

```
$ python -m venv  
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]  
           [--upgrade] [--without-pip] [--prompt PROMPT]  
           ENV_DIR [ENV_DIR ...]  
venv: error: the following arguments are required: ENV_DIR ①
```

- ① O **venv** exige pelo menos um parâmetro que é o diretório que manterá o ambiente virtual. Vamos criar um chamado **.venv** abaixo do diretório atual.

```
$ python -m venv .venv
```

Foi criada uma estrutura para manter o **site-packages**, totalmente isolado do seu sistema. O **site-packages** que está em **.venv/lib/python3.7/site-packages** (*o número da versão do Python pode variar*), começa praticamente vazio, normalmente tem apenas o **pip** e **setuptools** para dar suporte a instalação dos novos pacotes.

Porém o ambiente virtual ainda não está ativado (o que é necessário para sua real utilização), vejamos:

```
$ python -c "import site; print(site.getsitepackages()[0])"
```

```
/usr/lib/python3.7/site-packages ①
```

- ① O **site-packages** continua no caminho global do sistema

Aqui vemos que o ambiente global ainda está ativo, para ativar o ambiente virtual:

16.1.1. Ativando o ambiente virtual

```
$ source .venv/bin/activate  
$ echo $VIRTUAL_ENV
```

```
<diretório atual>/venv ①
```

- ① A variável de ambiente `$VIRTUAL_ENV` estará setada quando um ambiente estiver ativo, e terá o *path* absoluto do mesmo.

O comando de ativação muda conforme o sistema operacional e o *shell* utilizado, conforme a tabela abaixo.

Tabela 5. Comandos para ativação do venv conforme ambiente

Plataforma	Shell	Comando para ativação
Posix	bash/zsh	\$ source <venv>/bin/activate
	fish	\$. <venv>/bin/activate.fish
	csh/tcsh	\$ source <venv>/bin/activate.csh
Windows	cmd.exe	C:><venv>\Scripts\activate.bat
	PowerShell	PS C:><venv>\Scripts\Activate.ps1



Alguns *shells* já assumem a indicação do ambiente virtual no *prompt*.

Agora vamos reavaliar o nosso novo ambiente:

```
$ python -c 'import sys; print("\n".join(sys.path))'  
  
/usr/lib/python37.zip  
/usr/lib/python3.7  
/usr/lib/python3.7/lib-dynload  
<diretório atual>/venv/lib/python3.7/site-packages ①
```

- ① Podemos ver agora que o **site-packages** global foi substituído pelo caminho absoluto do **site-packages** do nosso ambiente virtual.

```
$ pip list  
  
Package      Version  
-----  
pip          19.0.3 ①  
setuptools   40.8.0 ①  
  
$ pip freeze ②
```

- ① A saída do `list` indica os pacotes `pip` e `setuptools`
② O `freeze` ignora estes pacotes e não retorna nenhum pacote para ser congelado

16.1.2. Procurando pacotes no PyPI

```
$ pip search requests  
  
requests (2.22.0)           - Python HTTP for Humans. ①
```

- ① Vários pacotes foram encontradas, mas listamos aqui apenas o que vamos utilizar

O sub-comando `search` do `pip`, executa uma busca no <https://pypi.org> (*Python Package Index*) e lista todos os pacotes encontrados, vamos instalar o `requests`, mas vamos supor que por algum motivo precisamos de uma versão específica deste pacote, digamos a 2.19.1, infelizmente não há nenhuma forma fácil de listar as versões de pacote, você pode acessar a página de histórico no pypi (<https://pypi.org/project/requests/#history>) ou forçar a instalação de uma versão não existente e o próprio erro lista as versões disponíveis.

16.1.3. Instalando um pacote de forma isolada

```
$ pip install requests==2.19.1 ①  
  
« instalação dos pacotes »  
  
Successfully installed certifi-2019.6.16 chardet-3.0.4 idna-2.7 requests-2.19.1 urllib3-1.23  
  
$ pip list ②  
Package      Version  
-----  
certifi      2019.6.16  
chardet     3.0.4  
idna        2.7  
pip          19.0.3  
requests    2.19.1  
setuptools   40.8.0  
urllib3     1.23  
  
$ pip freeze ③  
certifi==2019.6.16  
chardet==3.0.4  
idna==2.7  
requests==2.19.1  
urllib3==1.23
```

- ① Usando a mesma sintaxe do `freeze` podemos indicar precisamente a versão desejada
② Lista de pacotes instalados no ambiente virtual
③ Idem a lista, mas no formato do `freeze` e ignorando pacotes como `pip` e `setuptools`

16.1.4. Desativação

Como vimos temos um isolamento bastante preciso do nosso ambiente com o uso do `venv`. Vamos agora desativa-lo:

```
$ deactivate  
$ echo $VIRTUAL_ENV ①
```

- ① A variável `$VIRTUAL_ENV` não está mais setada

Habilidades adquiridas 🎓

- Biblioteca padrão (*baterias incluídas*):

`venv`

Módulo para gerir ambientes virtuais;

- Variáveis de ambiente:

`VIRTUAL_ENV`

Caminho do ambiente virtual ativado;

- Casos de uso do `pip`:

`search`

Localiza um pacote no PyPI;

`install <pacote>==<versao>`

Instala uma versão específica de um pacote;

- Comandos do ambiente virtual:

`deactivate`

Desativa o ambiente virtual.

Recursos externos 🔗

`venv` — *Creation of virtual environments* 🔗

<https://docs.python.org/3/library/venv.html>

Tutorial `venv` — *Virtual Environments and Packages* 🔗

<https://docs.python.org/3/tutorial/venv.html>

PEP 405 — *Python Virtual Environments*

<https://www.python.org/dev/peps/pep-0405>

`Virtualenv`

<https://virtualenv.pypa.io>

17. Banco de dados

17.1. PEP 249 — Python Database API Specification v2.0

Esta PEP define um conjunto básico padrão de *interfaces* (não formais) a serem seguidas para termos uma normalização entre os diversos pacotes disponíveis para acesso a banco de dados.

Nem todos os pacotes disponíveis suportam completamente esta especificação, mas se portabilidade for um requisito, este suporte se torna essencial.

Durante todo o curso tentaremos nos manter nesta API e exceções serão reportadas. Com isso a simples troca do banco e/ou pacote de acesso a dados não trará muitas mudanças, a exceção seriam pacotes que não atendem completamente este padrão e os *scripts SQL* que podem ter especificidades do banco de dados em uso.

17.2. Preparação do ambiente

Exercício 110 - Requerimentos para acesso a banco de dados

requerimentos.py

```
try:  
    from mysql import connector # noqa ① ②  
except ModuleNotFoundError: ③  
    print('MySQL Connector não instalado!')  
else:  
    print('MySQL Connector instalado e pronto para ser usado!') ④
```

① Pacote `mysql.connector` é necessário para acesso ao MySQL e é aderente ao PEP 249

② Um comentário `noqa` indica que os *linters* devem ignorar essa linha, neste caso é importante por que o ideal é fazer os *imports* no inicio do módulo

③ Ao tentar importar um módulo que não esteja instalado (ou não exista mesmo) levanta uma exceção `ModuleNotFoundError`

④ Um importação com sucesso indica existência do pacote necessário `mysql.connector`

Em um primeiro momento, o mais provável é que o `mysql.connector` não esteja instalado, retornando: `MySQL Connector não instalado!`. Com isso precisamos primeiro começar pela instalação do pacote e testar novamente, ver [Gerenciamento de pacotes](#).

```
$ pip install mysql.connector  
$ python requerimentos.py
```



Opcionalmente podemos criar um ambiente virtual para isolamento dos pacotes utilizados nessa seção. Ver [Isolamento de Ambientes](#).

17.3. Configuração do acesso ao banco de dados

Agora que já temos o pacote `mysql.connector` instalado, precisamos de um servidor **MySQL** ou **MariaDB**, precisaremos de um usuário e senha, para fins de testes assumimos usuário `root` com a senha `cod3r`, mas qualquer outro pode ser usado, desde que tenho direitos suficientes para criação de um banco de dados.

Caso possua uma instalação **docker** funcional, você pode subir um servidor com o comando abaixo:



```
docker container run --rm -p 3306:3306 -e MYSQL_ROOT_PASSWORD=cod3r -d mariadb:10.4
```

Importante: Neste curso não há suporte para o **docker**, listado aqui apenas como uma alternativa.

Exercício 111 - Configurando o acesso ao servidor

`configuracao.py`

```
from mysql.connector import connect

conexao = connect( ①
    host='localhost',
    port=3306,
    user='root',
    passwd='cod3r'
)
print(conexao) ②
```

① A função `connect()` do módulo `mysql.connector` aceita vários parâmetros nomeados, os mais importantes são: `host`, `port`, `user` e `password`

② Representação do objeto de conexão com o **MySQL**

17.4. Criação do nosso banco de dados

Agora vamos criar o banco de dados `agenda`, o qual usaremos nos próximos exercícios.

Exercício 112 - Criação do nosso banco de dados

criar_banco.py

```
from mysql.connector import connect

conexao = connect( ①
    host='localhost',
    port=3306,
    user='root',
    passwd='cod3r'
)
cursor = conexao.cursor() ②
cursor.execute('CREATE DATABASE agenda') ③
```

- ① O objeto de conexão retornado é uma instância da classe MySQLConnection
- ② O método MySQLConnection.cursor() retorna um “cursor” para ler e executar comandos no servidor de banco de dados, este objeto é uma instância da classe MySQLCursor. Este objeto é uma abstração especificada no DB-API 2.0 do Python. Isso nos dá a capacidade de ter vários ambientes de trabalho separados por meio da mesma conexão com o banco de dados
- ③ O método MySQLCursor.execute(), envia um comando SQL para ser executado no servidor, neste caso será um comando para criação do nosso banco de dados: CREATE DATABASE

Agora vamos conferir se o nosso banco de dados foi criado.

Exercício 113 - Listar banco de dados no servidor

listar_bancos.py

```
from mysql.connector import connect

conexao = connect(
    host='localhost',
    port=3306,
    user='root',
    passwd='cod3r'
)
cursor = conexao.cursor()
cursor.execute('SHOW DATABASES') ①

for i, database in enumerate(cursor, start=1): ② ③
    print(f'Banco de Dados {i}: {database[0]}') ④
```

① Comando para listar os banco de dados existentes no servidor: SHOW DATABASES

② Já estudamos antes o `enumerate()`, mas aqui vemos que ele possui um segundo argumento `start`, que permite indicar o valor inicial da contagem

③ O MySQLCursor suporta o método `next()`, ou seja um *iterator*, que se comporta de maneira similar ao `fetchone()` que veremos adiante, e portanto pode ser iterado das mais diversas formas — <https://www.python.org/dev/peps/pep-0249/#next>

④ O resultado do `MySQLCursor.execute()` é uma lista (*um item para cada registro*) de listas (*cada campo sendo um item, mesmo que tenha apenas um campo*), neste caso `database[0]` retornará o primeiro (e único) campo de cada registro

17.5. Uso do nosso banco de dados: agenda

Vamos agora configurar um acesso direto ao nosso banco de dados, e como isso será repetido durante vários exercícios vamos colocar esta configuração em um módulo a parte: `bd`.

Exercício 114 - Configuração do acesso ao banco agenda

bd.py

```
from contextlib import contextmanager
from mysql.connector import connect

parametros = dict( ①
    host='localhost',
    port=3306,
    user='root',
    passwd='cod3r',
    database='agenda' ②
)

@contextmanager ③
def nova_conexao():
    conexao = connect(**parametros)
    try:
        yield conexao ④
    finally:
        if (conexao and conexao.is_connected()): ⑤
            conexao.close() ⑥
```

- ① A variável `parametros` terá um dicionário que será usado nos próximos exercícios na conexão com o servidor
- ② Aqui temos mais um argumento disponível no `mysql.connector.connect()`, `database` indica um banco para ser selecionado ao conectar no servidor
- ③ O `contextlib.contextmanager` é um decorador que permite uma função ser usada no `with...as`
- ④ O `yield` permite o retorno parcial de uma função (usada em *generators*), e neste caso retorna o valor para o `with...as`
- ⑤ Verifica se a conexão foi criada e está ativa
- ⑥ Fecha a conexão

Agora teremos um exemplo de uso dos parâmetros definidos no módulo `utils`.

Exercício 115 - Conexão ao banco de dados: agenda

conexao.py

```
from bd import nova_conexao ①

with nova_conexao() as conexao: ②
    if conexao.is_connected(): ③
        print('Conectado ao banco de dados da agenda')
```

- ① Importação da função `nova_conexao()` a partir do módulo `bd`
- ② Conexão ao servidor, recebendo o resultado na variável `conexao`
- ③ Novo método `MySQLConnection.is_connected()` que retorna um `boolean` indicando se está conectado ou não ao servidor

Exercício 116 - Criação de tabelas

criar_tabela.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError ①
from bd import parametros

tabela_contatos = 'CREATE TABLE contatos(nome VARCHAR(50), tel VARCHAR(40))' ②
tabela_emails = 'CREATE TABLE emails(id INT AUTO_INCREMENT PRIMARY KEY, dono VARCHAR(50))' ③

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()

    cursor.execute(tabela_contatos) ④
    cursor.execute(tabela_emails) ④
except ProgrammingError as e: ⑤
    print(f'Erro: {e.msg}') ⑤
```

- ① Exceção `ProgrammingError` levantada pelo `mysql.connector` em várias situações
- ② SQL para criação de uma tabela `contatos` sem chave primária: `CREATE TABLE`
- ③ SQL para criação de uma tabela `emails`, já com chave primária e auto incremento: `CREATE TABLE`
- ④ Execução dos `scripts` SQL de criação de tabelas
- ⑤ Propriedade `ProgrammingError.msg` que contém a mensagem do erro levantado

Exercício 117 - Exclusão de tabela

excluir_tabela.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.execute('DROP TABLE emails') ①
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
```

- ① Execução do script SQL para exclusão da tabela de e-mails: DROP TABLE

Exercício 118 - Listar tabelas

listar_tabelas.py

```
from mysql.connector import connect
from bd import parametros

conexao = connect(**parametros)
cursor = conexao.cursor()
cursor.execute('SHOW TABLES') ①

for i, database in enumerate(cursor, start=1):
    print(f'Tabela {i}: {database[0]}'')
```

- ① Execução do script SQL para listar as tabelas: SHOW TABLES

Vamos melhorar nossa tabela `contatos` adicionando uma chave-primária.

Exercício 119 - Alterar tabela

alterar_tabela.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = 'ALTER TABLE contatos ADD COLUMN id INT AUTO_INCREMENT PRIMARY KEY' ①

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.execute(sql)
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
```

- ① Script SQL para adicionar uma chave primária de auto-incremento: ALTER TABLE

17.6. Manipulação de dados

Exercício 120 - Incluir contato

incluir_contato.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = 'INSERT INTO contatos (nome, tel) VALUES (%s, %s)' ①
args = ('Lucas', '123') ②

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.execute(sql, args) ③
    conexao.commit() ④
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    print('1 registro incluído, ID:', cursor.lastrowid) ⑤
```

- ① Script SQL para inserção de dados na tabela dos contatos: `INSERT INTO`
- ② Aqui estão os dados que vão ser inseridos, na mesma ordem das `%s` no `INSERT`
- ③ O método `MySQLCursor.execute()` possui um segundo argumento para a substituição segura de dados no *script*, evitando *SQL Injection*
- ④ O método `MySQLConnection.commit()` confirma as alterações feitas e as aplica no servidor, sem ele os dados não serão gravados
- ⑤ A propriedade `MySQLCursor.lastrowid` retorna o último `id` gerado no “cursor”

Exercício 121 - Incluir vários contatos

incluir_varios_contatos.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = 'INSERT INTO contatos (nome, tel) VALUES (%s, %s)'
contatos = (
    ('Arthur', '456'),
    ('Paulo', '789'),
    ('Ângelo', '000'),
    ('Eduardo', '987'),
    ('Yuri', '654'),
    ('Leonardo', '321'),
)
try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.executemany(sql, contatos) ❶
    conexao.commit()
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    print(f'Foram adicionados {cursor.rowcount} registros') ❷
```

❶ O método MySQLCursor.executemany() executa o *script* passado no primeiro argumento para cada iteração do segundo argumento, que pode ser uma lista por exemplo.

❷ A propriedade MySQLCursor.rowcount retorna o número de registros afetados pela última execução (neste caso o executemany)

17.7. Seleção de dados

Exercício 122 - Selecionar contatos

selecionar_contatos.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = 'SELECT * FROM contatos' ①

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.execute(sql)
    contatos = cursor.fetchall() ②
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    for contato in contatos:
        print(f'{contato[2]:2d} - {contato[0]:35s} Telefone: {contato[1]}') ③
```

- ① Selecionar todos os campos e todos os registros da tabela de contatos: SELECT
- ② O método MySQLCursor.fetchall() carrega todos os registros disponíveis e retorna uma lista
- ③ Acesso aos campos na mesma ordem que se encontram no tabela

Podemos ser mais precisos, selecionando exatamente que campos precisamos e em que ordem, ajudando inclusive a otimizar uso do banco de dados, rede e/ou memória.

Exercício 123 - Selecionar campos

selecionar_campos.py

```
from mysql.connector import connect
from bd import parametros

conexao = connect(**parametros)
cursor = conexao.cursor()
cursor.execute('SELECT nome, tel FROM contatos') ①

for registro in cursor.fetchall():
    print('\t'.join(str(campo) for campo in registro))
```

- ① Aqui temos a especificação de uma lista de campos e não mais um *

Existem situações em que precisamos apenas do primeiro registro, ou desejamos consumir os dados aos poucos.

Exercício 124 - Selecionar um registro

selecionar_um_registro.py

```
from mysql.connector import connect
from bd import parametros

conexao = connect(**parametros)
cursor = conexao.cursor()
cursor.execute('SELECT * FROM contatos')
print(cursor.fetchone()) ①
```

- ① O método `MySQLCursor.fetchone()` carrega o próximo registro indicado no “cursor”, neste caso o primeiro. Deixando-o pronto para a próxima solicitação



Este recurso é o mesmo que foi usado na [Exercício 118 - Listar tabelas](#), em que iteramos diretamente no “cursor”. Ele é preferível em relação ao `fetchall()` já que se comporta como um *stream*, e utiliza apenas o recurso necessário pelo tempo necessário, e não em lote.

Exercício 125 - Selecionar com filtro

selecionar_com_filtro.py

```
from mysql.connector import connect
from bd import parametros

sql = "SELECT * FROM contatos WHERE tel = '456'" ①

conexao = connect(**parametros)
cursor = conexao.cursor()
cursor.execute(sql)

for x in cursor: ②
    print(x)
```

- ① Script SQL filtrando apenas quando o tel for igual a 456: `SELECT...WHERE`
② Iterando diretamente no “cursor” é como executar `fetchone()` até não ter mais registros para recuperar

Exercício 126 - Selecionar com filtro parcial

selecionar_filtro_parcial.py

```
from mysql.connector import connect
from bd import parametros

sql = "SELECT * FROM contatos WHERE nome LIKE '%ng%'" ①

conexao = connect(**parametros)
cursor = conexao.cursor()
cursor.execute(sql)

for x in cursor:
    print(x)
```

- ① Script SQL filtrando apenas quando o nome contiver a string 'ng': SELECT...WHERE...LIKE

Exercício 127 - Filtrar a parte de entrada do usuário

selecionar_sem_sql_injection.py

```
from mysql.connector import connect
from bd import parametros

nome = input('Contato a localizar: ')

sql = 'SELECT * FROM contatos WHERE nome LIKE %s'
args = (f'%{nome}%', ) ①

conexao = connect(**parametros)
cursor = conexao.cursor()
cursor.execute(sql, args)

for x in cursor:
    print(x)
```

- ① O uso dos argumentos no MySQLCursor.execute() se torna essencial para higienizar dados oriundos de fontes externas, como entrada do usuário.

Exercício 128 - Ordenar contatos

ordenar_contatos.py

```
from mysql.connector import connect
from bd import parametros

sql = 'SELECT nome FROM contatos ORDER BY nome' ①

conexao = connect(**parametros)
cursor = conexao.cursor()
cursor.execute(sql)

print('\n'.join(registro[0] for registro in cursor))
```

① Script SQL ordenando os contatos pelo nome: SELECT...ORDER BY



Podemos ordenar de forma decrescente acrescentando a cláusula DESC após o ORDER BY

```
sql = 'SELECT nome FROM contatos ORDER BY nome DESC'
```

Exercício 129 - Excluir contato

excluir_contato.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = 'DELETE FROM contatos WHERE nome = %s' ①
args = ('Yuri',)

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.execute(sql, args)
    conexao.commit()
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    print(f'{cursor.rowcount} registro(s) deletado(s).')
```

① Script SQL para exclusão de dados: DELETE FROM...WHERE...

Exercício 130 - Atualizar contato

atualizar_contato.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = 'UPDATE contatos SET nome = %s WHERE nome = %s' ①
args = ('Galdino', 'Lucas')

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.execute(sql, args)
    conexao.commit()
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    print(f'{cursor.rowcount} registro(s) deletado(s).')
```

- ① Script SQL para atualização de dados: UPDATE...SET...WHERE...

Exercício 131 - Limitar de registros

selecionar_com_limite.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = 'SELECT * FROM contatos ORDER BY nome LIMIT 3' ①

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.execute(sql)
    contatos = cursor.fetchall() ②
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    for contato in contatos:
        print(f'{contato[2]:2d} - {contato[0]:35s} Telefone: {contato[1]}')
```

- ① A cláusula LIMIT indica a quantidade máxima de registros que serão recuperadas pelo SELECT

- ② Apesar do uso preferencial do consumo dos registros sob demanda (*iterator* ou *fetchone()*), em casos como este, em que há um limite definido, o uso *fetchall()* pode ser a melhor opção



A cláusula LIMIT não faz parte da especificação SQL ANSI, portanto pode não existir em outros servidores de banco de dados, ou possuir outra sintaxe.

Exercício 132 - Limitar de registros com offset

selecionar_com_limite_offset.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = 'SELECT * FROM contatos ORDER BY nome LIMIT 3 OFFSET 2' ①

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.execute(sql)
    contatos = cursor.fetchall()
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    for contato in contatos:
        print(f'{contato[2]:2d} - {contato[0]:35s} Telefone: {contato[1]}')
```

- ① A cláusula `LIMIT...OFFSET` faz com que o `SELECT` pule os registros iniciais, antes de começar a selecionar os registros, podemos traduzir: `LIMIT 3 OFFSET 2` para “a partir do terceiro registro (pulando o *offset*) retorne 3 registros”.



Assim como a cláusula `LIMIT`, o `OFFSET` não faz parte da especificação SQL ANSI. Além de não poder ser usada isoladamente do `LIMIT`.

17.8. Associação

Vamos preparar uma nova tabela para testarmos as associações. Com ela vamos agrupar os contatos.

Exercício 133 - Criar tabela de grupos

criar_grupo.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

tabela_grupo = 'CREATE TABLE grupos (id INT AUTO_INCREMENT PRIMARY KEY, descricao VARCHAR(30))'
contato_grupo = 'ALTER TABLE contatos ADD grupo_id INT, ADD FOREIGN KEY (grupo_id) REFERENCES grupos (id)'

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.execute(tabela_grupo)
    cursor.execute(contato_grupo)
    conexao.commit()
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
```

Exercício 134 - Povoar tabela de grupos

povoar_grupo.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = 'INSERT INTO grupos (descricao) VALUES (%s)'
grupos = (
    ('Casa',),
    ('Trabalho',),
)

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()
    cursor.executemany(sql, grupos)
    conexao.commit()
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    print(f'{cursor.rowcount} grupos adicionados')
```

Exercício 135 - Atualizar contatos com os grupos

associar_grupo_contato.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

selecionar_grupo = 'SELECT id FROM grupos WHERE descricao = %s'
atualizar_contato = 'UPDATE contatos SET grupo_id = %s WHERE nome = %s'
contato_grupo = {
    'Arthur': 'Casa',
    'Eduardo': 'Trabalho',
    'Lucas': 'Trabalho',
    'Paulo': 'Casa',
}

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor()

    for contato, grupo in contato_grupo.items():
        cursor.execute(selecionar_grupo, (grupo,))
        grupo_id = cursor.fetchone()[0]
        cursor.execute(atualizar_contato, (grupo_id, contato))

    conexao.commit()
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    print('contatos associados')
```

Exercício 136 - Associar contatos e grupos

contatos_com_grupo.py

```
from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = """SELECT grupos.descricao AS grupo, contatos.nome AS contato
         FROM contatos
        INNER JOIN grupos ON contatos.grupo_id = grupos.id ORDER BY grupo, contato"""\n\n①

try:
    conexao = connect(**parametros)
    cursor = conexao.cursor(dictionary=True) ②
    cursor.execute(sql)
    contatos = cursor.fetchall()
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    for contato in contatos:
        print(f'{contato["grupo"]}: {contato["contato"]}')
```

- ① A cláusula INNER JOIN...ON permite a associação entre duas tabelas através da expressão definida no ON
- ② No package `mysql.connector`, o método `cursor()`, aceita um parâmetro nomeado `Dictionary`, sendo passado um argumento verdadeiro (`True`), muda o resultado da execução para uma lista de dicionários, ao invés de lista de listas.



Este recurso do dicionário não faz parte do PEP 249, e pode não existir em outros *packages*, ou ter sintaxe diferente.

Exercício 137 - Associar contatos e grupos — versão 2

contatos_com_grupo_v2.py

```
from collections import defaultdict ①

from mysql.connector import connect
from mysql.connector.errors import ProgrammingError
from bd import parametros

sql = """SELECT grupos.descricao AS grupo, contatos.nome AS contato
         FROM contatos
        INNER JOIN grupos ON contatos.grupo_id = grupos.id
        ORDER BY grupo, contato"""

try:
    conexao = connect(**parametros)
    try: ②
        cursor = conexao.cursor(dictionary=True)
        try: ③
            cursor.execute(sql)
            contatos = cursor.fetchall()
        finally:
            cursor.close() ③
    finally:
        conexao.close() ②

    print(f'Conectado? {conexao.is_connected()}') ④
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
else:
    agrupados = defaultdict(list) ⑤
    for contato in contatos:
        agrupados[contato['grupo']].append(contato['contato']) ⑤

    print(agrupados) ⑥
    print('\n'.join(f'{grupo}: {" ".join(contatos)}'
                   for grupo, contatos in agrupados.items()))
```

- ① O módulo `collections` da biblioteca padrão, possui uma classe `defaultdict` que é uma especialização do `dict`, e permite definir um valor *default* para chaves ainda não existentes
- ② `MySQLConnection.close()` finaliza a conexão com o servidor, liberando recursos, em um script pode não ser tão essencial já que este método é chamado assim que o objeto é destruído (se a conexão ainda estiver ativa)
- ③ `MySQLCursor.close()` finaliza o “cursor”, liberando recursos. Este também é chamado assim que o objeto é destruído (e o “cursor” ainda esteja ativo)
- ④ Confere se a conexão foi finalizada
- ⑤ Ao inicializar um `defaultdict` é necessário passar um *callable* que fornecerá o valor padrão, neste caso acessos a chave não existentes retornam uma lista vazia e permite o método `list.append()`
- ⑥ Representação do `defaultdict`

17.9. SQLite

SQLite é um biblioteca em C que implementa um motor de banco de dados leve, rápido, alto contido, compatível com SQL ANSI e pode ser embutido nas mais diversas aplicações, por conta disso foi incorporada a biblioteca padrão Python a partir da versão 2.5.

Devido a sua alta portabilidade e tamanho diminuto, este banco de dados é muito usado para armazenamento local de dados relacionais.

Exercício 138 - Uso do SQLite

sqlite.py

```
from sqlite3 import connect, ProgrammingError, Row ①

tabela_grupo = 'CREATE TABLE IF NOT EXISTS grupos (id INTEGER PRIMARY KEY AUTOINCREMENT, descricao VARCHAR(30))' ②
tabela_contatos = """CREATE TABLE IF NOT EXISTS contatos
    (id INTEGER PRIMARY KEY AUTOINCREMENT,
     nome VARCHAR(50),
     tel VARCHAR(40),
     grupo_id INTEGER)"""
insert_grupos = 'INSERT INTO grupos (descricao) VALUES (?)' ③
select_grupos = 'SELECT id, descricao FROM grupos'
insert_contatos = 'INSERT INTO contatos (nome, tel, grupo_id) VALUES (?, ?, ?)'
select = """SELECT grupos.descricao AS grupo, contatos.nome AS contato
    FROM contatos
    INNER JOIN grupos ON contatos.grupo_id = grupos.id
    ORDER BY grupo, contato"""

try:
    conexao = connect(':memory:') ④
    conexao.row_factory = Row ⑤
    cursor = conexao.cursor()

    cursor.execute(tabela_grupo)
    cursor.execute(tabela_contatos)

    cursor.executemany(insert_grupos, (('Casa',), ('Trabalho',)))
    cursor.execute(select_grupos)
    grupos = {row['descricao']: row['id'] for row in cursor.fetchall()} ⑥

    contatos = (
        ('Arthur', '456', grupos['Casa']), ⑦
        ('Paulo', '789', grupos['Casa']),
        ('Ângelo', '000', grupos['Trabalho']),
        ('Eduardo', '987', None),
        ('Yuri', '654', None),
        ('Leonardo', '321', None),
    )
    cursor.executemany(insert_contatos, contatos)

    cursor.execute(select)
    for contato in cursor:
        print(contato['contato'], contato['grupo']) ⑧
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
```

- ① Os *imports* necessários variam conforme o *package* utilizado, neste caso da biblioteca padrão: `sqlite3`
- ② A cláusula `IF NOT EXISTS` faz com que a tabela só seja criada, se ainda não existir. Apesar de ser prática, não faz parte do SQL ANSI e portanto nem todos os servidores de banco de dados a implementam, e mesmo quando tem podem não ter a mesma sintaxe
- ③ Os estilos de parâmetros variam conforme biblioteca, no `sqlite3` é a `?`, mas pode ser modificada
- ④ O primeiro argumento do `sqlite3.connect` é o nome do arquivo aonde o banco será gravado, podendo ser `:memory:`, o que indica que será um banco em memória (perdido no final da execução)
- ⑤ A propriedade `sqlite3.Connection.row_factory` permite definirmos um *callable* para

transformar a *list* retornada pelos *fetches* em outro objeto como um *dict*, neste caso o `sqlite3.Row` já faz este trabalho em um objeto que é similar a um *dict*

- ⑥ Transformação da lista de dicionários em um dicionário simples {*descricao*: *id*, ...}
- ⑦ Usando o *dict* grupos para transformar a descrição do grupo no *id*
- ⑧ Consumindo como uma lista de dicionários



Veja que a forma de recuperar os dados como um dicionário no `sqlite3` é bem diferente do `mysql.connector`, exatamente por não ser definido no PEP 249.

Habilidades adquiridas

- Biblioteca padrão (*baterias incluídas*):

`collections`

Pacote com funções e classes referentes a coleção;

`defaultdict`

Classe especializada de *dict* que permite um valor *default*, para chaves não encontradas;

`sqlite3`

Módulo para acesso ao banco de dados SQLite;

`sqlite3.Connection.row_factory`

Callable para transformar os dados lidos do “cursor”;

`sqlite3.Row`

Callable para o `row_factory`, que cria um *dict like* com o nome dos campos;

- Pacotes de terceiros:

`mysql.connector`

Módulo para acesso ao banco de dados MySQL;

- PEP 249:

`connection.connect()`

Abrir uma conexão com o servidor de banco de dados;

`connection.close()`

Fecha o cursor, ocorre automaticamente na destruição do objeto;

`connection.commit()`

Aplica as mudanças no servidor;

```
cursor.execute()  
    Executa um script SQL;  
  
cursor.executemany()  
    Executa um script SQL para cada iteração dos argumentos passados;  
  
cursor.fetchone()  
    Recupera o próximo registro de um “cursor”;  
  
cursor.fetchall()  
    Recupera todos os registros em uma lista;  
  
cursor.close()  
    Fecha o cursor, ocorre automaticamente na destruição do objeto;  
  
cursor.rowcount  
    Número de registros afetados pela última execução de script SQL;  
  
cursor.lastrowid  
    Último id gerado pelo “cursor”;  
  
ProgrammingError  
    Exceção principal na execução de scripts;
```

Recursos externos

PEP 249 — *Python Database API Specification v2.0*

<https://www.python.org/dev/peps/pep-0249>

sqlite3 — DB-API 2.0 *interface for SQLite databases* 

<https://docs.python.org/3/library/sqlite3.html>

MySQL Connector for Python

<https://dev.mysql.com/downloads/connector/python>

Pacotes alternativos para acesso ao MySQL

<https://wiki.python.org/moin/MySQL>

SQLite

<https://www.sqlite.org>

Desafio

Fazer uma lista de tarefas (*to do list*) com persistência.

Funcionalidades

- Adicionar tarefa

- Listar tarefas pendentes (todas são adicionadas como pendentes)
- Concluir tarefa

Sugestão de algoritmo

1. Listar as tarefas pendentes numeradas
2. Perguntar por nova tarefa, número da tarefa a concluir ou **sair**
3. Caso texto digitado seja igual a **sair**, finaliza a execução
4. Caso digitado um número de tarefa válido, atualiza-la como concluída e voltar ao **passo 1**
5. Caso não, adicionar tarefa e voltar ao **passo 1**

Exemplo de solução disponível em [Desafio 11 - Lista de tarefas com persistência](#)

Quero mais

Reescrever o desafio acima em outro paradigma, seja estruturado, funcional ou orientado a 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)))
```

MDC

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}{" " if attrs else ""}{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=None):
        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
```

Lista de tarefas persistente

Desafio 11 - Lista de tarefas com persistência

desafio_db.py

```
import sys
from sqlite3 import connect, ProgrammingError, Row

tabela = 'CREATE TABLE IF NOT EXISTS todo (id INTEGER PRIMARY KEY AUTOINCREMENT, descricao VARCHAR(30), feito boolean)'
pendentes = 'SELECT id, descricao FROM todo WHERE not feito'
adicionar = 'INSERT INTO todo (descricao, feito) VALUES (?, false)'
concluir = 'UPDATE todo SET feito = true WHERE id = ?'

try:
    conexao = connect('todo.db')
    conexao.row_factory = Row
    cursor = conexao.cursor()
    cursor.execute(tabela)

    while True:
        cursor.execute(pendentes)
        tarefas = cursor.fetchall()

        for i, tarefa in enumerate(tarefas, start=1):
            print(f'{i} - {tarefa["descricao"]}')

        print()
        print('Indique o número de uma tarefa para concluir, digite uma nova tarefa ou "sair":')
        entrada = input(':')

        if entrada.lower() == 'sair':
            sys.exit(0)

        if entrada.isnumeric() and int(entrada) in range(1, len(tarefas) + 1):
            cursor.execute(concluir, (tarefas[int(entrada)-1]['id'],))
        elif entrada.strip() and not entrada.isnumeric():
            cursor.execute(adicionar, (entrada.strip(),))
        else:
            print('> Entrada inválida!')

        conexao.commit()
except ProgrammingError as e:
    print(f'Erro: {e.msg}')
```

Anexo B: Exemplos avançados

Fibonacci

Exemplo Avançado 1 - Fibonacci recursivo decrescente sem *memoize*

ex-fibonacci_recursive_decrecente.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('http://www.geoservicos.ibge.gov.br/geoserver/wms?service=WFS&version=1.0.0&request=GetFeature&typeName=C GEO:Rede UrbanaSintese_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](#)

[Equivalência entre operadores × métodos no set](#)

[Comandos para ativação do venv conforme ambiente](#)

Listas de figuras e diagramas

[Diagrama das classes de manipulação de arquivo](#)

[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 - Conjuntos](#)

[Exercício 17 - Conjunto \(operações\)](#)

[Exercício 18 - Interpolações](#)

[Exercício 19 - Área do círculo - versão 1](#)

[Exercício 20 - Área do círculo - versão 2](#)

[Exercício 21 - Área do círculo - versão 3](#)

[Exercício 22 - Área do círculo - versão 4](#)

[Exercício 23 - Área do círculo - versão 5](#)

[Exercício 24 - Área do círculo - versão 5 \(correção\)](#)

[Exercício 25 - Área do círculo - versão 6](#)

[Exercício 26 - Área do círculo - versão 7](#)

[Exercício 27 - Área do círculo - versão 8](#)

[Exercício 28 - Área do círculo - versão 9](#)

[Exercício 29 - Área do círculo - versão 10](#)

[Exercício 30 - Área do círculo - versão 11](#)

[Exercício 31 - Área do círculo - versão 12](#)

[Exercício 32 - Área do círculo - versão 13](#)

[Exercício 33 - Área do círculo - versão 14](#)

[Exercício 34 - Fibonacci - While infinito](#)

[Exercício 35 - Fibonacci - While condicional](#)

[Exercício 36 - Fibonacci - Uso do *packing*](#)

[Exercício 37 - Fibonacci - Iterando uma lista](#)

[Exercício 38 - Fibonacci - sum](#)

[Exercício 39 - Fibonacci - break](#)

[Exercício 40 - Fibonacci - range\(\)](#)

[Exercício 41 - Fibonacci recursivo](#)

[Exercício 42 - Fibonacci recursivo com operador ternário](#)

[Exercício 43 - Leitura Básica de Arquivo](#)

[Exercício 44 - Leitura Básica de Arquivo \(*stream*\)](#)

[Exercício 45 - Leitura Básica de Arquivo \(*stream*\) — Fix](#)

[Exercício 46 - Mais robustez com try...finally](#)

[Exercício 47 - Leitura de Arquivo com with](#)

[Exercício 48 - Gravação de Arquivo](#)

[Exercício 49 - Leitura de Arquivo com o módulo csv](#)

[Exercício 50 - Dobros](#)

[Exercício 51 - Dobros dos pares](#)

[Exercício 52 - Generators](#)

[Exercício 53 - Generators com for](#)

[Exercício 54 - Dict Comprehension](#)

[Exercício 55 - Funções de primeira classe](#)

[Exercício 56 - Funções de alta ordem](#)

[Exercício 57 - Funções com escopos aninhados \(*closure*\)](#)

[Exercício 58 - Totalização de compras \(`lambda`\)](#)

[Exercício 59 - Totalização de compras sem o uso de `lambda`](#)

[Exercício 60 - Cálculo de fatorial usando recursividade](#)

[Exercício 61 - Listar todos os meses do ano com 31 dias](#)

[Exercício 62 - Listar todos os meses do ano com 31 dias \(funcional em uma linha\)](#)

[Exercício 63 - Listar todos os meses do ano com 31 dias \(imperativo\)](#)

[Exercício 64 - Diversas funções úteis trabalham com objetos imutáveis](#)

[Exercício 67 - Implementação do *generator map*](#)

[Exercício 68 - Implementação do `map` com *generator expression*](#)

[Exercício 69 - Parâmetros opcionais \(com valores *default*\)](#)

[Exercício 70 - Argumentos nomeados](#)

[Exercício 71 - Unpacking de argumentos](#)

[Exercício 72 - Combinando unpacking e parâmetros opcionais](#)

[Exercício 73 - Unpacking de argumentos nomeados](#)

[Exercício 74 - Objetos chamáveis](#)

[Exercício 75 - Problemas com argumentos mutáveis ☠](#)

[Exercício 76 - Problemas com argumentos mutáveis \(solução\)](#)

[Exercício 77 - Decorator log](#)

[Exercício 78 - Conhecendo o elif](#)

[Exercício 79 - Simulando um switch](#)

[Exercício 80 - Switch com valores únicos](#)

[Exercício 81 - Switch baseado em faixa de valores](#)

[Exercício 82 - Conhecendo a fundo o while](#)

[Exercício 83 - Conhecendo a fundo o for](#)

[Exercício 84 - Uso de variável de controle extra no for](#)

[Exercício 85 - Uso do else no for](#)

[Exercício 86 - Conhecendo a fundo o try](#)

[Exercício 87 - Execução de uma função em outro package](#)

[Exercício 88 - Momento de execução do código](#)

[Exercício 89 - Uso de módulos com mesmo nome](#)

[Exercício 90 - Importação direta das funções no namespace atual](#)

[Exercício 91 - Uso de um pacote como façade](#)

[Exercício 92 - Consumidor do pacote do desafio](#)

[Exercício 93 - Classe Task](#)

[Exercício 94 - Classe Project](#)

[Exercício 95 - Método __iter__\(\)](#)

[Exercício 96 - Implementação do vencimento \(datetime e timedelta\)](#)

[Exercício 97 - Herança](#)

[Exercício 98 - Métodos “privados” e simulação de “overload”](#)

[Exercício 99 - Sobrecarga de operador](#)

[Exercício 100 - Snake trap - Tratamento de exceções](#)

[Exercício 101 - Membros de classe × membros da instância](#)

[Exercício 102 - Tipos de métodos](#)

[Exercício 103 - Propriedades através de métodos](#)

[Exercício 104 - Utilizando o *decorator* @property](#)

[Exercício 105 - Método abstrato usando NotImplementedError](#)

[Exercício 106 - Método abstrato usando o módulo abc: *Abstract Base Class*](#)

[Exercício 107 - Herança múltipla](#)

[Exercício 108 - Mixins](#)

[Exercício 109 - Iterator](#)

[Exercício 110 - Requerimentos para acesso a banco de dados](#)

[Exercício 111 - Configurando o acesso ao servidor](#)

[Exercício 112 - Criação do nosso banco de dados](#)

[Exercício 113 - Listar banco de dados no servidor](#)

[Exercício 114 - Configuração do acesso ao banco agenda](#)

[Exercício 115 - Conexão ao banco de dados: agenda](#)

[Exercício 116 - Criação de tabelas](#)

[Exercício 117 - Exclusão de tabela](#)

[Exercício 118 - Listar tabelas](#)

[Exercício 119 - Alterar tabela](#)

[Exercício 120 - Incluir contato](#)

[Exercício 121 - Incluir vários contatos](#)

[Exercício 122 - Selecionar contatos](#)

[Exercício 123 - Selecionar campos](#)

[Exercício 124 - Selecionar um registro](#)

[Exercício 125 - Selecionar com filtro](#)

[Exercício 126 - Selecionar com filtro parcial](#)

[Exercício 127 - Filtrar a parte de entrada do usuário](#)

[Exercício 128 - Ordenar contatos](#)

[Exercício 129 - Excluir contato](#)

[Exercício 130 - Atualizar contato](#)

[Exercício 131 - Limitar de registros](#)

[Exercício 132 - Limitar de registros com offset](#)

[Exercício 133 - Criar tabela de grupos](#)

[Exercício 134 - Povoar tabela de grupos](#)

[Exercício 135 - Atualizar contatos com os grupos](#)

[Exercício 136 - Associar contatos e grupos](#)

[Exercício 137 - Associar contatos e grupos — versão 2](#)

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](#)

[Desafio 11 - Lista de tarefas com persistência](#)

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

Argumento

Valores passados para uma chamada de função, método ou criação de classe.

Garbage Colector

Processo de liberar a memória, quando os objetos não são mais necessários (ou referenciados).

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](#)

Módulo

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

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 ofidioglotas conhecidos são descendentes de Salazar Sonserina, com **Harry Potter** sendo uma notável exceção. — <http://pt-br.harrypotter.wikia.com/wiki/Ofidioglossia>

Pacote

É um conjunto de módulos, que pode ser interno ou externo, como aqueles gerenciados pelo `pip` e normalmente publicados no [PyPi](#).

Parâmetro

Identificadores nas funções e métodos que receberão os argumentos passados.

PyPi

Python Package Index — Diretório *online* de pacotes públicos para **Python**.

Zen of Python

Criado por Tim Peters, representa os alicerces fundamentais da linguagem. Pode ser lido através do comando: `import this`

Importante conhecer



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