

MANUAL DO FORMANDO

Criar aplicações em linguagem de programação Python

Manual do Formando

Ação de formação	UC00618 — Criar aplicações em linguagem de programação Python (50 horas · 4,5 pontos de crédito)
Modalidade / Regime	Formação Modular — Ativos Empregados · Formação a Distância (Microsoft Teams)
Datas	08/10/2026 a 30/11/2026 · 2.ª e 5.ª feiras, 18:00–21:00
Formador	Luís Ferreira
Entidade formadora	IEFP, I.P. — Centro de Emprego da Maia
Contactos	Rua Dr. Carlos Felgueiras, 418 · 4470-157 Maia · Tel.: 220 989 190 · Email: ce.maia@iefp.pt
Referencial	https://catalogo.snq.gov.pt/ucDetalhe/355650



Cofinanciado pela
União Europeia

Índice de conteúdos

Módulo 1. Pensamento computacional e algoritmos

Módulo 2. Python e ambiente de desenvolvimento (Anaconda, VS Code, Spyder, Jupyter)

Módulo 3. Fundamentos da linguagem: tipos de dados, coleções e controlo do programa

Módulo 4. Funções, ficheiros, iteradores e geradores

Módulo 5. Programação orientada a objetos

Módulo 6. Tratamento de dados: texto, CSV, Excel e SQL (Pandas e NumPy)

Módulo 7. Análise gráfica e bibliotecas: Matplotlib, Turtle/Pygame e widgets

Módulo 8. Testes, depuração, desempenho e documentação

Introdução

Este manual acompanha a ação de formação «Criar aplicações em linguagem de programação Python» (UC00618) e serve de apoio durante e após as sessões. Cada capítulo corresponde a um módulo do programa e combina a explicação dos conceitos com exemplos de código prontos a experimentar e dicas de utilização.

Convenções usadas neste manual:

- Os blocos de código estão em letra monoespaçada e podem ser copiados para o VS Code ou para o Jupyter.
- As linhas que começam por >>> representam a consola interativa do Python (REPL); as restantes são ficheiros .py.
- Os comentários (# ...) explicam o código e são ignorados pelo interpretador.
- Os exemplos foram testados com Python 3.14.7. Funcionam em qualquer versão 3.10 ou superior (a instrução match exige 3.10+).



Dica: Experimente sempre o código. Programar aprende-se a escrever, a errar e a corrigir — ler não chega.

Como está organizado o curso

1. Módulo 1 — Pensamento computacional e algoritmos
2. Módulo 2 — Python e ambiente de desenvolvimento (Anaconda, VS Code, Spyder, Jupyter)
3. Módulo 3 — Fundamentos da linguagem: tipos de dados, coleções e controlo do programa
4. Módulo 4 — Funções, ficheiros, iteradores e geradores
5. Módulo 5 — Programação orientada a objetos
6. Módulo 6 — Tratamento de dados: texto, CSV, Excel e SQL (Pandas e NumPy)
7. Módulo 7 — Análise gráfica e bibliotecas: Matplotlib, Turtle/Pygame e widgets
8. Módulo 8 — Testes, depuração, desempenho e documentação

Módulo 1 · Pensamento computacional e algoritmos

Pensamento computacional

Pensamento computacional é a capacidade de formular problemas de modo a que possam ser resolvidos por um computador (ou por uma pessoa a seguir instruções). Assenta em quatro princípios:

- Decomposição — dividir um problema grande em partes mais pequenas e tratáveis.
- Reconhecimento de padrões — identificar semelhanças e repetições entre problemas.
- Abstração — concentrar-se no essencial e ignorar os detalhes irrelevantes.
- Algoritmos — definir a sequência de passos que resolve o problema.

Algoritmos

Um algoritmo é uma sequência finita, ordenada e não ambígua de passos que resolve um problema. Pode representar-se em linguagem natural, em pseudocódigo ou em fluxograma.

```
ALGORITMO média_de_notas
  LER nota1, nota2, nota3
  media <- (nota1 + nota2 + nota3) / 3
  SE media >= 9.5 ENTÃO
    ESCRIVER "Aprovado"
  SENÃO
    ESCRIVER "Reprovado"
  FIM SE
FIM
```

Pseudocódigo: as três estruturas básicas — sequência, decisão (SE) e, noutros casos, repetição (ENQUANTO/PARA).

Num fluxograma: o oval marca início/fim, o retângulo uma ação, o paralelogramo entrada/saída e o losango uma decisão com dois caminhos (sim/não).

Metodologias de desenvolvimento de software

Criar um programa não é só escrever código. As metodologias organizam o trabalho em fases:

9. Análise — perceber o problema e os requisitos (o que o programa tem de fazer).
10. Desenho — planejar a solução: dados, estrutura, algoritmos, interface.
11. Implementação — escrever o código.
12. Testes — verificar que o programa faz o que deve e corrigir erros.
13. Manutenção — corrigir, adaptar e melhorar ao longo do tempo.

No modelo em cascata as fases seguem-se em sequência; nas metodologias ágeis (Scrum, Kanban) trabalha-se em ciclos curtos, entregando versões funcionais e incorporando feedback. Neste curso usaremos uma abordagem iterativa: pequenos programas que crescem a cada sessão.



Dica: Antes de escrever código, escreva o algoritmo em 5 a 10 linhas de pseudocódigo. Poupa tempo e erros.

Módulo 2 · Python e ambiente de desenvolvimento

A linguagem Python

Python é uma linguagem de alto nível, interpretada, de sintaxe clara e com uma enorme biblioteca de módulos. É usada em automação, análise de dados, ciência, web, inteligência artificial, ensino e administração de sistemas.

- Vantagens: legibilidade, produtividade, multiplataforma, comunidade e bibliotecas (pandas, numpy, matplotlib...).
- Versões: a linha 2.x terminou em 2020; use sempre Python 3. A versão 3.14 (2025/2026) trouxe melhorias de desempenho, mensagens de erro mais claras e o novo gestor de instalação para Windows.
- Interpretada: o interpretador executa o código linha a linha — ideal para experimentar na consola (REPL).

```
>>> 2 + 3 * 4
14
>>> "Olá, " + "Maia"
'Olá, Maia'
>>> print("Python", 3.14)
Python 3.14
```

Consola interativa: escreva python no terminal e experimente.

Ambientes de desenvolvimento

- Visual Studio Code — editor leve e extensível; com a extensão Python torna-se um IDE completo (execução, depuração, testes, notebooks). É o ambiente principal do curso.
- Anaconda — distribuição que inclui Python, centenas de bibliotecas científicas e o gestor conda. Traz o Spyder (IDE orientado à análise de dados, com explorador de variáveis) e o Jupyter.
- Jupyter Notebook — documentos interativos com células de código e de texto (Markdown); excelentes para explorar dados e documentar o raciocínio. O VS Code abre notebooks nativamente.

Funcionalidades do editor de texto

Um bom editor ajuda a escrever código correto: realce de sintaxe, indentação automática, autocompletar (IntelliSense), formatação automática, deteção de erros em tempo real, pesquisa/substituição e integração com o terminal e com o Git.

O primeiro script

```
# ola.py – o meu primeiro programa
nome = input("Como se chama? ")
print(f"Olá, {nome}! Bem-vindo(a) ao Python.")
```

Guarde como ola.py e execute no terminal: python ola.py

 **Dica:** Consulte o «Guia de Instalação do Ambiente» (material 17) para instalar o Python 3.14.7 e o VS Code passo a passo, e a página de apoio do curso para tirar dúvidas.

Módulo 3 · Fundamentos da linguagem

Variáveis e tipos de dados

Uma variável é um nome que referencia um valor. Em Python não se declara o tipo — ele é inferido do valor.

```
idade = 34           # int
preco = 19.99        # float
nome = "Ana"         # str
ativo = True         # bool
print(type(preco))   # <class 'float'>
total = preco * 3     # 59.97
print(int("42") + 1) # conversão de tipo: 43
```

Operadores

- Aritméticos: + - * / // (divisão inteira) % (resto) ** (potência).
- Comparação: == != < <= > >= → devolvem True/False.
- Lógicos: and, or, not.

```
>>> 17 // 5, 17 % 5, 2 ** 10
(3, 2, 1024)
>>> 10 > 3 and not (2 == 3)
True
```

Strings

```
s = "Programação em Python"
print(len(s))           # 22
print(s[0], s[-1])      # P n
print(s[0:12])           # Programação (fatiamento: início:fim)
print(s.upper(), s.lower().count("o"))
print(s.replace("Python", "Python 3.14"))
print(s.split(" "))      # ['Programação', 'em', 'Python']
n, v = "Ana", 1234.5
print(f"{n} pagou {v:.2f} €") # f-string com formatação
```

Listas, tuplos, dicionários e conjuntos

list notas = [12, 15, 9]

Coleção ordenada e mutável. Métodos: append, insert, remove, pop, sort, reverse, index, count.

tuple ponto = (3, 4)

Coleção ordenada e imutável. Útil para dados fixos e para devolver vários valores de uma função.

dict aluno = {"nome": "Rui", "nota": 14}

Pares chave → valor, sem duplicados na chave. Métodos: keys, values, items, get, pop, update.

```
set cores = {"azul", "verde"}
```

Conjunto sem repetidos, não ordenado. Operações: união |, interseção &, diferença -.

```
notas = [12, 15.5, 9, 18]
notas.append(14)
print(sum(notas) / len(notas), max(notas), sorted(notas, reverse=True))
positivas = [n for n in notas if n >= 9.5]      # compreensão de lista
agenda = {"Ana": "912000000", "Rui": "934000000"}
agenda["Marta"] = "961000000"
for nome, tel in agenda.items():
    print(nome, tel)
print(agenda.get("Zé", "não existe"))
```

Controlo do programa: if / elif / else e match

```
nota = float(input("Nota: "))
if nota >= 17:
    print("Muito bom")
elif nota >= 14:
    print("Bom")
elif nota >= 9.5:
    print("Suficiente")
else:
    print("Insuficiente")

opcao = input("Opção (1-Somar, 2-Subtrair, 0-Sair): ")
match opcao:
    case "1": print("Somar")
    case "2": print("Subtrair")
    case "0": print("Adeus")
    case _:  print("Opção inválida")
```

A indentação (4 espaços) define os blocos — não é opcional em Python.

Ciclos: for e while

```
for i in range(1, 11):      # 1 ... 10
    print(f"7 x {i} = {7 * i}")

for indice, nome in enumerate(["Ana", "Rui"], start=1):
    print(indice, nome)

import random
segredo = random.randint(1, 100)
tentativas = 0
while True:
    palpite = int(input("Palpite: "))
    tentativas += 1
    if palpite == segredo:
        print(f"Acertou em {tentativas} tentativas!")
        break
    print("Maior" if palpite < segredo else "Menor")
```

 **Dica:** Siga a PEP 8 (guia de estilo oficial): nomes em snake_case, 4 espaços de indentação, linhas até 79/99 caracteres, espaços à volta dos operadores. O VS Code formata automaticamente com Shift+Alt+F.

Módulo 4 · Funções, ficheiros, iteradores e geradores

Funções

```
def area_circulo(raio: float) -> float:
    """Devolve a área de um círculo com o raio indicado."""
    import math
    return math.pi * raio ** 2

def saudacao(nome, cumprimento="Olá"):          # parâmetro por omissão
    return f"{cumprimento}, {nome}!"

def media(*valores):                             # n.º variável de argumentos
    return sum(valores) / len(valores)

print(area_circulo(2), saudacao("Ana"), saudacao("Rui", cumprimento="Bom dia"), media(10,
12, 14))
dobro = lambda x: x * 2                          # função anónima
print(list(map(dobro, [1, 2, 3])), list(filter(lambda n: n % 2 == 0, range(10))))
Docstrings ("""" ... """) documentam a função; as anotações de tipo (: float, -> float) ajudam o editor e o leitor.
```

Âmbito: variáveis criadas dentro de uma função são locais. Evite variáveis globais — passe valores por parâmetro e devolva resultados com return.

Módulos e pacotes

```
import math, random
from datetime import date, timedelta
print(math.sqrt(16), random.choice(["cara", "coroa"]), date.today() + timedelta(days=7))

# módulo próprio: utilitarios.py na mesma pasta
from utilitarios import area_circulo
# pacotes externos: no terminal -> pip install requests
A biblioteca padrão traz centenas de módulos (os, sys, json, csv, re, pathlib, sqlite3...).
```

Ficheiros e exceções

```
from pathlib import Path

caminho = Path("dados") / "servidor.log"
try:
    with open(caminho, encoding="utf-8") as f:          # with fecha o ficheiro automaticamente
        for linha in f:                                # lê linha a linha
            if "ERROR" in linha:
                print(linha.strip())
except FileNotFoundError:
    print("Ficheiro não encontrado:", caminho)

with open("relatorio.txt", "w", encoding="utf-8") as out: # "w" escreve, "a" acrescenta
    out.write("Relatório gerado\n")
Modos: r (ler), w (escrever), a (acrescentar); use sempre encoding="utf-8".
```

Iteradores e geradores

Um iterador produz valores um de cada vez (protocolo iter/next). Um gerador é a forma mais simples de criar iteradores: uma função com yield, que «pausa» e retoma — ideal para ficheiros grandes, porque não carrega tudo em memória.

```
def pares(limite):
```

```
n = 0
while n <= limite:
    yield n
    n += 2

def ler_linhas(caminho):
    with open(caminho, encoding="utf-8") as f:
        for linha in f:
            yield linha.rstrip("\n")

print(list(pares(10)))                # [0, 2, 4, 6, 8, 10]
quadrados = (x * x for x in range(1_000_000))  # expressão geradora: nada é
calculado ainda
print(sum(quadrados))
erros = sum(1 for l in ler_linhas("dados/servidor.log") if "ERROR" in l)
```

 **Dica:** Sempre que escrever o mesmo código duas vezes, transforme-o numa função. Sempre que uma lista for enorme e só a percorrer uma vez, use um gerador.

Módulo 5 · Programação orientada a objetos

Na programação orientada a objetos (POO) organizamos o programa em objetos que juntam dados (atributos) e comportamentos (métodos). Uma classe é o molde; um objeto é uma instância desse molde.

Classes, objetos, atributos e métodos

```
class ContaBancaria:
    taxa_manutencao = 0.5          # atributo de classe (comum a todas)

    def __init__(self, titular, saldo=0.0):  # construtor
        self.titular = titular             # atributos de instância
        self.saldo = saldo

    def depositar(self, valor):
        self.saldo += valor

    def levantar(self, valor):
        if valor > self.saldo:
            raise ValueError("Saldo insuficiente")
        self.saldo -= valor

    def __str__(self):              # representação para print()
        return f"Conta de {self.titular}: {self.saldo:.2f} €"

conta = ContaBancaria("Ana", 100)
conta.depositar(50)
conta.levantar(30)
print(conta)                      # Conta de Ana: 120.00 €
```

self é a referência ao próprio objeto; é sempre o primeiro parâmetro dos métodos.

Propriedades e métodos especiais

```
class Produto:
    def __init__(self, nome, preco):
        self.nome = nome
        self._preco = preco          # convenção: _ indica «interno»

    @property
    def preco(self):
        return self._preco

    @preco.setter
    def preco(self, valor):
        if valor < 0:
            raise ValueError("Preço negativo")
        self._preco = valor

    def __repr__(self):
        return f"Produto({self.nome!r}, {self._preco})"

    def __eq__(self, outro):
        return self.nome == outro.nome
```

Herança e polimorfismo

```
class Veiculo:
    def __init__(self, marca, modelo):
```

```

        self.marca, self.modelo = marca, modelo
    def descricao(self):
        return f"{self.marca} {self.modelo}"

class Carro(Veiculo):
    def __init__(self, marca, modelo, portas):
        super().__init__(marca, modelo)          # chama o construtor da classe base
        self.portas = portas
    def descricao(self):                          # sobreposição
        return super().descricao() + f" ({self.portas} portas)"

class Moto(Veiculo):
    def descricao(self):
        return "Moto " + super().descricao()

for v in [Carro("Renault", "Clio", 5), Moto("Honda", "CB500")]:
    print(v.descricao())                        # polimorfismo: o mesmo método,
comportamentos diferentes

```

Decoradores e dataclasses

```

import time
from functools import wraps
from dataclasses import dataclass

def cronometro(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        inicio = time.perf_counter()
        resultado = func(*args, **kwargs)
        print(f"{func.__name__} demorou {time.perf_counter() - inicio:.4f} s")
        return resultado
    return wrapper

@cronometro
def soma_até(n):
    return sum(range(n + 1))

@dataclass
class Cliente:                                # gera __init__, __repr__, __eq__
    nome: str
    email: str
    pontos: int = 0

    @classmethod
    def anonimo(cls):
        return cls("Anónimo", "n/d")

    @staticmethod
    def valida_email(email):
        return "@" in email

print(soma_até(1_000_000), Cliente("Ana", "ana@ex.pt"), Cliente.anonimo())

```

 **Dica:** Use POO quando há entidades com estado e comportamento (Cliente, Encomenda, Jogo). Para scripts pequenos, funções chegam. As dataclasses poupam muito código repetitivo.

Módulo 6 · Tratamento de dados: texto, CSV, Excel e SQL

Ficheiros de texto e CSV (biblioteca padrão)

```
import csv
totais = {}
with open("dados/vendas.csv", newline="", encoding="utf-8") as f:
    for linha in csv.DictReader(f, delimiter=";"):
        totais[linha["regiao"]] = totais.get(linha["regiao"], 0) + float(linha["valor"])
print(totais)
```

NumPy

```
import numpy as np
a = np.array([3, 7, 1, 9])
print(a * 2, a.mean(), a.max(), a[a > 2])      # operações vetoriais, sem ciclos
m = np.arange(12).reshape(3, 4)               # matriz 3x4
print(m.sum(axis=0))
```

Pandas: DataFrames

```
import pandas as pd
df = pd.read_csv("dados/vendas.csv", sep=";")      # também:
pd.read_excel("dados/clientes.xlsx")
print(df.head(), df.info(), df.describe())
print(df[df["regiao"] == "Norte"])                 # filtrar linhas
df["valor_iva"] = df["valor"] * 1.23               # coluna calculada
resumo = df.groupby("regiao")["valor"].sum().sort_values(ascending=False)
pivot = df.pivot_table(values="valor", index="regiao", columns="mes", aggfunc="sum")
df["valor"] = df["valor"].fillna(0)                # valores em falta
resumo.to_excel("resumo_vendas.xlsx")              # exportar (precisa de openpyxl)
df.to_csv("vendas_tratadas.csv", index=False, sep=";")
```

Instalar: `pip install pandas openpyxl`

SQL com sqlite3 e Pandas

```
import sqlite3, pandas as pd
con = sqlite3.connect("dados/loja.sqlite")
cur = con.cursor()
cur.execute("""CREATE TABLE IF NOT EXISTS clientes(
    id INTEGER PRIMARY KEY, nome TEXT, cidade TEXT)""")
cur.execute("INSERT INTO clientes(nome, cidade) VALUES (?, ?)", ("Ana", "Maia"))
con.commit()
for linha in cur.execute("SELECT * FROM clientes WHERE cidade = ?", ("Maia",)):
    print(linha)
df = pd.read_sql("""SELECT c.nome, SUM(e.quantidade * p.preco) AS total
    FROM encomendas e
    JOIN clientes c ON c.id = e.cliente_id
    JOIN produtos p ON p.id = e.produto_id
    GROUP BY c.nome ORDER BY total DESC""", con)
con.close()
```

Use sempre parâmetros (?) nas consultas — nunca concatene texto do utilizador em SQL.

 **Dica:** Fluxo típico de um projeto de dados: importar → inspecionar → limpar → transformar → agregar → exportar/visualizar.

Módulo 7 · Análise gráfica e bibliotecas

Matplotlib

```
import matplotlib.pyplot as plt
import pandas as pd
df = pd.read_csv("dados/vendas.csv", sep=";")
por_mes = df.groupby("mes", sort=False)["valor"].sum()

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
ax1.bar(por_mes.index, por_mes.values, color="#007a38")
ax1.set_title("Vendas por mês"); ax1.set_ylabel("€")
df.groupby("regiao")["valor"].sum().plot.pie(ax=ax2, autopct="%1.0f%%", title="Por região")
plt.tight_layout()
plt.savefig("vendas.png", dpi=150)
plt.show()
```

Tipos: plot (linhas), bar/barh, scatter, pie, hist. Personalize títulos, eixos, legendas, cores e grelha.

Turtle

```
import turtle
t = turtle.Turtle(); t.speed(0)
def poligono(n, lado):
    for _ in range(n):
        t.forward(lado); t.left(360 / n)
for i in range(12):
    poligono(6, 80); t.left(30)
turtle.done()
```

«flor» de hexágonos

Pygame (jogos)

```
import pygame
pygame.init()
ecra = pygame.display.set_mode((640, 400)); relógio = pygame.time.Clock()
x, y, vx, vy = 320, 200, 4, 3
a_correr = True
while a_correr:
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT: a_correr = False
    x += vx; y += vy
    if x < 15 or x > 625: vx = -vx
    if y < 15 or y > 385: vy = -vy
    ecra.fill((10, 40, 20)); pygame.draw.circle(ecra, (102, 196, 48), (x, y), 15)
    pygame.display.flip(); relógio.tick(60)
pygame.quit()
```

pip install pygame

Widgets: tkinter e ipywidgets

```
import tkinter as tk
def converter():
    c = float(entrada.get()); resultado.config(text=f"{c * 9 / 5 + 32:.1f} °F")
janela = tk.Tk(); janela.title("Conversor")
entrada = tk.Entry(janela); entrada.pack(padx=10, pady=5)
tk.Button(janela, text="Converter", command=converter).pack()
resultado = tk.Label(janela, text="-"); resultado.pack(pady=5)
janela.mainloop()
```

```
# Em Jupyter: pip install ipywidgets
# from ipywidgets import interact
# interact(lambda n: n ** 2, n=(0, 10))
```

 **Dica:** Um gráfico deve responder a uma pergunta. Título claro, eixos identificados, poucas cores e sem efeitos 3D.

Módulo 8 · Testes, depuração, desempenho e documentação

Testes unitários

```
# test_utilitarios.py
import unittest
from utilitarios import area_circulo, e_primo

class TestUtilitarios(unittest.TestCase):
    def test_area_zero(self):
        self.assertEqual(area_circulo(0), 0)
    def test_area(self):
        self.assertAlmostEqual(area_circulo(1), 3.14159, places=4)
    def test_primos(self):
        self.assertTrue(e_primo(7)); self.assertFalse(e_primo(1))
    def test_raio_negativo(self):
        with self.assertRaises(ValueError):
            area_circulo(-1)

if __name__ == "__main__":
    unittest.main()
```

Executar: `python -m unittest -v` ou `pytest -v` (pip install pytest). O VS Code tem um painel de testes.

Depuração

- Leia o traceback de baixo para cima: a última linha diz o tipo de erro; as anteriores, onde ocorreu.
- Depurador do VS Code: coloque um breakpoint (clique à esquerda do número da linha), prima F5, avance com F10 (passo) / F11 (entrar), inspecione variáveis no painel lateral e no «Watch».
- logging em vez de print: `import logging; logging.basicConfig(level=logging.INFO); logging.info("...")`.

```
Traceback (most recent call last):
  File "bugs.py", line 12, in <module>
    print(lista[5])
IndexError: list index out of range
```

Erros comuns: `SyntaxError`, `NameError`, `TypeError`, `ValueError`, `IndexError`, `KeyError`, `ZeroDivisionError`, `FileNotFoundError`.

Benchmark e profilers

```
import timeit, cProfile, tracemalloc
print(timeit.timeit("sum(x*x for x in range(10_000))", number=100))
print(timeit.timeit("[x*x for x in range(10_000)]", number=100))

def trabalho():
    return sorted(str(i) for i in range(200_000))
cProfile.run("trabalho()", sort="cumulative")          # perfil de CPU: onde se gasta o tempo

tracemalloc.start()
dados = trabalho()
atual, pico = tracemalloc.get_traced_memory()          # perfil de memória
print(f"pico: {pico / 1e6:.1f} MB"); tracemalloc.stop()
```

Meça antes de otimizar. A maior parte do tempo está em 5% do código.

Documentação e normas

- Docstrings em módulos, classes e funções; README.md com objetivo, instalação, utilização e autor.

- Guião técnico (arquitetura, dependências, decisões), relatório (resultados, testes) e manual de utilizador (passo a passo com imagens).
- Normas e regulamentos: PEP 8 (estilo), PEP 257 (docstrings), licenças de software (MIT, GPL...), RGPD no tratamento de dados pessoais, regulamento do formando e política de utilização dos sistemas da entidade.
- Controlo de versões com Git/GitHub: guarde o histórico e partilhe o projeto.

 **Dica:** Um programa sem testes e sem documentação não está terminado. Reserve 20% do tempo do projeto final para ambos.

Notas finais e recursos

Este manual é um ponto de partida. A melhor forma de consolidar os conhecimentos é resolver os exercícios fornecidos e aplicar as técnicas a dados e problemas reais do seu contexto profissional.

- Documentação oficial: <https://docs.python.org/3/> (tutorial em <https://docs.python.org/3/tutorial/>)
- Pandas: <https://pandas.pydata.org/docs/> · Matplotlib: <https://matplotlib.org/stable/>
- VS Code + Python: <https://code.visualstudio.com/docs/python/python-tutorial>
- PEP 8: <https://peps.python.org/pep-0008/>
- Exercícios: <https://exercism.org/tracks/python> · <https://www.w3schools.com/python/>

Bom trabalho!