NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO DE
INFORMÁTICA

DEPARTMENT OF
COMPUTER SCIENCE

Informática para Ciências e Engenharias

Projeto de Programação (v1.1) — 2023/24

Equipa de ICE 2023-24

13 de Maio de 2024

1 Objetivo do Trabalho

Os isolantes térmicos são muito usados em aplicações industriais, construção civil, transportes e equipamento de proteção. Estudar o desempenho de materiais isolantes é importante para seleccionar os mais adequados a cada aplicação.

Assumiremos que a transferência de calor entre um material aquecido a uma temperatura inicial T_0 e mantido num ambiente arrefecido a 0°C segue uma curva exponencial inversa, dada pela lei de Newton para o arrefecimento, que permite calcular a temperatura T_t a cada instante t:

$$T_t = T_0 \cdot e^{-k \cdot t} \tag{1}$$

onde k é o coeficiente de transferência térmica, t é o tempo e T_0 é a temperatura inicial.

O objectivo deste trabalho é criar um programa que, através da utilização de uma base de dados, facilite a organização e armazenamento de dados sobre a capacidade de isolamento térmico de vários materiais, bem como a consulta dessa base de dados para a produção de relatórios e gráficos. O programa deverá fazer uso das linguagens `Python` e `SQL`, e do sistema de base de dados `SQLite3`.

2 O Programa a Desenvolver

Pretende-se um programa cuja função principal terá a seguinte assinatura:

```
def process_cmds(db_file: str) -> None:
```

Os parâmetros da função `process_cmds()` são os seguintes:

`db_file` — contém o nome de uma base de dados *SQLite3*. Este parâmetro tem de conter sempre um nome válido como, por exemplo, "`dados.db`".

Campus de Caparica
2829-516 CAPARICA | Portugal
T: +351 212 948 536

O programa deve ler comandos do teclado e executá-los. Os comandos devem ser introduzidos um por linha e podem surgir por qualquer ordem.

São aceites os seguintes comandos:

CREATE_TABLES

```
def cmd_create_tables(db: sqlite3.Connection) -> None:
```

Ao encontrar este comando o programa deve apagar as duas tabelas da base de dados, caso existam. De seguida, deve criar as duas tabelas (vazias) na base de dados. As duas tabelas utilizadas por este programa são:

Tests — tabela que define um ensaio. Note que se assume que, em todos os ensaios, a temperatura ambiente era de 0°C, de forma a poder-se aplicar a Equação (1). Note que a informação acerca de qual material foi testado em cada ensaio é guardada nesta tabela `Tests` e não na tabela `Samples`.

Esquema da tabela `Tests`:

Campo	***Tipo***	**Descrição**
`id`	*um número inteiro*	Identificador do ensaio.
`mat_id`	*uma string com tamanho fixo de 6 caracteres*	A referência do material.
`year`	*um número inteiro*	Ano em que foi produzido o ensaio.
`temp_ini`	*um número real*	Temperatura inicial do material, quando foi realizada a amostra.
`certification`	*uma string com um tamanho máximo de 20 caracteres*	Certificação do laboratório (i.e., "`Certified`", "`Not Certified`", ou "`Unknown`").

Samples — tabela com a descrição das amostras realizadas durante um ensaio.

Esquema da tabela:

Campo	***Tipo***	**Descrição**
`id`	*um número inteiro*	Identificador da amostra.
`test_id`	*um número inteiro*	Identificador do ensaio onde foi obtido este resultado.
`time`	*um número inteiro*	Tempo que passou desde o início da experiência até que foi realizada a amostra.
`temperature`	*um número real*	Temperatura do material quando foi realizada a amostra.

LOAD_TEST file[;file;file;...]

```
def cmd_load_test(db: sqlite3.Connection,
                  args: str) -> None:
```

Ler os ficheiros com os nomes indicados e carregar a informação para as tabelas da base de dados `db_file`. Podem ser indicados um ou mais ficheiros, separados pelo caráter "`;`". Os parentesis retos significam *opcional* e não deverão ser incluídos no comando!

Cada ensaio é descrito num ficheiro que tem o seguinte formato:

1ª linha Código do ensaio;

2ª linha Ano do ensaio;

3ª linha Nome do material;

4ª lina Certificação do laboratório (e.g., `Certified`, "Not Certified", "Unknown");

5ª linha Temperatura inicial usada no ensaio;

Restantes linhas identificador de cada valor, o tempo que mediou desde o início do ensaio (em horas) e temperatura (em °C), separados pelo carácter "`;`".

Exemplo de um ficheiro a carregar:

```
4
2003
IUORBA
Not Certified
200
401;30;54.75
402;48;25.48
403;61;14.52
404;67;11.27
405;71;9.59
406;90;4.27
```

SUMMARY start;end;certification

```
def cmd_summary(db: sqlite3.Connection,
                args: str) -> None:
```

Escrever no ecrã as referências dos materiais para os quais há ensaios no período delimitado pelos valores `start` e `end` (inclusive) e feitos em laboratórios com a certificação especificada (`certification`). Estes três parâmetros estão separados por "`;`". Qualquer um dos três parâmetros pode ser substituído pelo carácter "`*`" que terá o seguinte significado:

`start == "*"` — deverá considerar a data de início como *a menor data de todos os ensaios existentes na base de dados*;

`end == "*"` — deverá considerar a data de fim como *a maior data de todos os ensaios existentes na base de dados*;

`certification == "*"` — deverá ignorar este campo no processo de filtragem (i.e., qualquer valore em `certificacao` será aceite);

Exemplo de comandos `SUMMARY`:

```
SUMMARY 2005;2010;*
SUMMARY *;*;*
SUMMARY 2006;*;Certified
```

O resultado de comandos `SUMMARY` deverá ter várias linhas:

1. Uma primeira linha indicando (**em Inglês**) quantos materiais satisfazem as condições impostas e quais eram estas condições;
2. De seguida deve haver uma linha por cada material que satisfaz as condições impostas, indicando a referência desse material (com um `"\t"` antes de cada referência).
3. Uma linha final indicando o número total de amostras de todos os materiais listados.

Este é o resultado esperado no ecrã para os três comandos acima, com os ficheiros de dados fornecidos juntamente com este enunciado.

```
6 materials between 2005 and 2010 width certification *
    AKAQQC
    JTKIHQ
    OJTLRM
    IKHCPB
    CQLKGJ
    PQQTCH
Total samples: 88

7 materials between * and * width certification *
    IUORBA
    AKAQQC
    PQQTCH
    CQLKGJ
    JTKIHQ
    OJTLRM
```

```
    IKHCPB
Total samples: 115

4 materials between 2006 and * width certification Certified
    OJTLRM
    JTKIHQ
    CQLKGJ
    PQQTCH
Total samples: 37
```

SUMMARY_FILE filename;start;end;certification

```
def cmd_summary_file(db: sqlite3.Connection,
                     args: str) -> None:
```

Este comando é em tudo idêntico ao anterior, com a exceção de que tem um parâmetro adicional "*filename*" (na primeira posição). O resultado do comando, em vez de ser apresentado para o ecrã, deverá ser escrito no ficheiro indicado.

PLOT ref[;ref;ref;...]

```
def cmd_plot(db: sqlite3.Connection,
             args: str) -> None:
```

Criar um gráfico com todos os pontos obtidos para o material indicado em `codigo`. Para cada ponto, a temperatura deve ser normalizada, dividindo-a pela temperatura inicial usada nesse ensaio, de forma a que todas as temperaturas fiquem numa escala de valores reais entre 0 a 1, indicando a fração da temperatura inicial (temperatura relativa). Deve também desenhar uma linha que representa o arrefecimento segundo a lei de Newton (Equação 1), estimando o valor da constante `k` para cada produto a partir das amostras existentes desse produto. Esta linha deverá resultar de uma interpolação de 1000 pontos unidos por uma linha. Podem ser indicados uma ou mais referências de materiais, separadas pelo caráter "`;`". Os parentesis retos significam *opcional* e não deverão ser incluídos no comando!

O valor da constante `k` pode ser estimado pela seguinte expressão:

$$k = \frac{1}{N} \sum_{n=1}^{N} \frac{-\log T_n}{t_n} \tag{2}$$

onde T_n é a temperatura relativa no tempo t_n. A tabela na Figura 1 ilustra a normalização das temperaturas para o material com a referência "`OJTLRM`" (nos ficheiros fornecidos), convertendo os valores para valores relativos à temperatura inicial em cada ensaio, e o gráfico do topo ilustra o resultado esperado. O outro gráfico, por baixo do primeiro, ilustra o resultado esperado quando é passado mais que uma referência ao comando `PLOT`.

Amostras para o material "OJTLRM"

Tempo	T	T_o	$T_{rel.}$
17	69.67	200	0.35
28	37.51	200	0.19
33	28.05	200	0.14
54	8.00	200	0.04
55	7.64	200	0.04
76	2.16	200	0.01
82	1.47	200	0.01
86	1.18	200	0.01
11	92.09	180	0.51
18	61.41	180	0.34
19	55.86	180	0.31
36	20.44	180	0.11
51	8.52	180	0.05
65	3.75	180	0.02
68	3.18	180	0.02
80	1.56	180	0.01
83	1.24	180	0.01
92	0.75	180	0.00

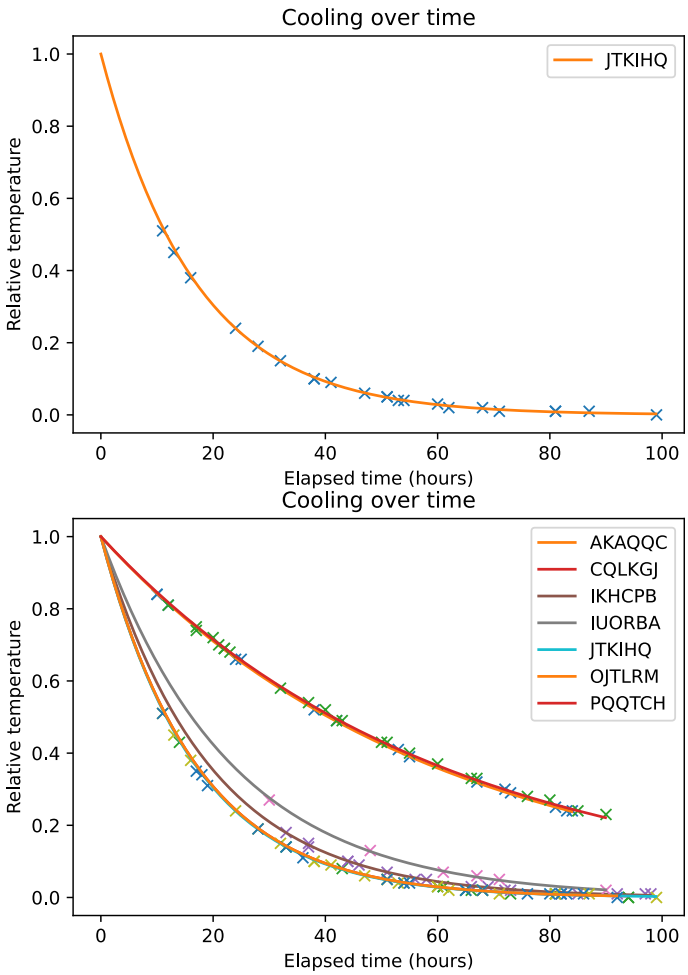

Figura 1: Exemplo de gráficos

PLOT_FILE filename;ref[;ref;ref;...]

```
def cmd_plot_file(db: sqlite3.Connection,
                  args: str) -> None:
```

Este comando é semelhante ao comando PLOT. No entanto, em vez de mostrar o gráfico gerado no ecrã, este comando guarda o gráfico num ficheiro com o nome dado (filename).

Exemplos:

```
PLOT_FILE amostas-1.jpg;IUORBA
PLOT_FILE amostas-2.png;IUORBA;AKAQQC
PLOT_FILE amostas-3.pdf;IUORBA;AKAQQC;PQQTCH;CQLKGJ;JTKIHQ;OJTLRM;IKHCPB
```

EXECUTE file

```
def cmd_execute(db: sqlite3.Connection,
                args: str) -> None:
```

Executa os comandos lidos do ficheiro "`filename`", um por linha, como se tivessem sido introduzidos a partir do teclado.

3 Ordem de Implementação dos Comandos

Deve implementar um comando de cada vez e testá-lo exaustivamente, de forma a garantir a correta implementação de cada comando antes de passar ao comando seguinte.

Sugerimos que implemente os comandos pela seguinte ordem:

1. `CREATE_TABLES`;
2. `LOAD_TEST` — só com um ficheiro;
3. `SUMMARRY` — sem "*";
4. `EXECUTE`;
5. `PLOT` — só com uma referência.

Depois de ter um programa com os comando acima a funcionar corretamente, melhore o seu program adicionando as seguintes funcionalidades:

1. Acrescente o suporte para os "*" no comando `SUMMARY`;
2. Acrescente a possiblidade de ter múltiplos ficheiros no comando `LOAD_TEST`;
3. Acrescente a possiblidade de ter múltiplas referências no comando `PLOT`;
4. Adicione a implementação do comando `SUMMARY_FILE`;
5. Adicione a implementação do comando `PLOT_FILE`;

4 Dados Experimentais para o Projeto

O arquivo `ice-2023-24-projeto.zip`, disponibilizado no CLIP, tem os ficheiros:

`orders_N.txt` — com `N` variando de 1 a 3, que contêm uma sequência de comandos com complexidade crescente;

`ficheiro_N.txt` om `N` variando de 1 a 15, com dados de ensaios que podem ser usados como exemplos para testar o seu programa.

NOTA: pode assumir que todas as sequências de comandos começam por criar as tabelas, depois carregar ensaios, e só depois realizar comandos variados (`SUMMARY`, `PLOT`, etc) sobre os ensaios. Para terminar o programa deverá utilizar o comando `QUIT`.

5 Entrega do Trabalho

Garanta que no início do programa (o ficheiro `.py` que entrega) tem um comentário com:

1. Os números e os nomes dos elementos do grupo;
2. Quais foram as principais contribuições de cada elemento do grupo (i.e., um resumo de o que é que cada um fez);
3. Como é que os elementos do grupo distribuem a quantidade de trabalho/contribuição de cada um, em percentagem. A soma das percentagens deve dar 100%. Em caso de discordância entre os elementos do grupo, colocar duas versões, uma para cada elemento do grupo. *Esta distribuição de trabalho/contribuição poderá diferenciar positiva e negativamente a classificação final do projeto para os dois elementos do grupo.*

O trabalho tem de ser entregue até às 23h59 do dia 6 de junho de 2023.

O trabalho é entregue no servidor `mooshak`, cujo endereço e processo de submissão será anunciado brevemente.

6 Critérios de Avaliação do Trabalho

De acordo com o Regulamento de Avaliação de Conhecimentos da FCT/UNL[1] os estudantes diretamente envolvidos numa fraude são liminarmente reprovados na disciplina. Em ICE, considera-se que um aluno que dá ou que recebe código num trabalho comete fraude. Igualmente, sendo o trabalho para realizar em grupos de um ou de dois alunos, considera-se que as situações em que os alunos realizam o trabalho em grupos maiores, partilhando o código, são situações de fraude equivalentes a dar e receber o código entre os vários grupos.

Os trabalhos serão alvo de uma comparação automática com um software de deteção de plágio/cópia. Situações indicadas pelo software como de potencial plágio serão alvo de análise pela equipa docente e, a confirmarem-se as evidências, levarão à reprovação à disciplina.

A pontuação obtida no site `mooshak` é apenas um indicador dos testes passados corretamente e não a nota do trabalho. Os trabalhos serão avaliados de acordo com critérios bem definidos, dos quais se salienta:

- Programação usando a linguagem Python:

[1] https://www.fct.unl.pt/sites/default/files/regulamento_avaliacao_revisao_julho_2020_versao_final_31-07-2020_alt1_04-11-2020_17-11-2020.pdf

  - Utilização correta dos elementos básicos da linguagem Python e regras de boa programação apresentadas e praticadas nas aulas;
  - Apenas pode usar os módulos usados nas práticas: `math`, `sqlite3`, `requests` e `matplotlib`;
  - Código legível (nomes, indentação, etc.) e comentado;
  - Correta utilização de *type hints*;
  - Correcta documentação das funções com *docstrings*.
- Desenvolvimento do programa:
  - Utilização correta das regras e boas práticas de programação apresentadas e praticadas nas aulas;
  - Decomposição adequada do problema em sub-problemas;
  - Implementação genérica (o programa deve ser capaz de processar diferentes dados e variadas sequências de comandos dados pelo utilizador e num ficheiro de comandos);
- Resolução do problema apresentado no enunciado:
  - Número de comandos implementados;
  - Qualidade, correção e completude da implementação de cada comando, como por exemplo:
    * Obtenção dos resumos (comando `SUMMARY`) corretos, de acordo com os parâmetros dados;
    * Desenho dos gráficos com os pontos corretos e a interpolação também;
    * Desenho do gráfico e apresentação dos resumos como exemplificados neste enunciado (por exemplo, o título nos resumos e o título e as legendas no gráfico);
  - Para as funções pedidas no enunciado, deve respeitar os nomes e argumentos para que passam ser chamadas (utilizadas) pelo docente tal e qual como indicadas nos exemplos.

A classificação final do trabalho será um número entre zero e vinte, arredondado às centésimas, dependendo dos comandos implementados corretamente e dos critérios anteriores de avaliação. Mesmo que não consiga implementar todos os comandos, é muito melhor implementar apensa alguns corretamente do que tentar implementar todos, mas nenhum deles funcionar bem. Também, a incorreção de uma função/comando pedida neste enunciado não deve impedir a programação e o teste das outras funções/comandos.

APÊNDICES

A Utilização de Ferramentas de IA em ICE

Atualmente, as ferramentas de inteligência artificial generativa têm sido cada vez mais integradas no apoio à programação, facilitando e democratizando o processo de desenvolvimento de software. Assim, é importante abordar e clarificar o tema da utilização das ferramentas de inteligência artificial (IA) no funcionamento da UC e, em especial, no desenvolvimento deste projeto final de programação.

As ferramentas de inteligência artificial generativa, como por exemplo o ChatGPT, são uma ferramentas poderosas que podem oferecer suporte no desenvolvimento de programas em Python. No entanto, é crucial reconhecer os possíveis desafios e limitações associados ao uso destas ferramentas. Tal como ocorre com o uso indiscriminado de material encontrado na Internet, a utilização dessas ferramentas acarreta riscos, podendo facilmente obter-se resultados incorretos ou mesmo totalmente errados/falsos.

Sem o devido conhecimento e discernimento, há uma elevada probabilidade de os estudantes utilizarem a ferramenta de maneira inadequada, comprometendo assim a integridade do trabalho académico e os objetivos educacionais propostos na UC de ICE.

Assim:

- A utilização não referenciada de material encontrado na Internet é considerada fraude académica e resulta na reprovação imediata à UC;
- Não é permitida a utilização de ChatGPT, nem de outras ferramentas similares de geração automática de código, no projeto final da Unidade Curricular (UC) de ICE, mesmo quando devidamente referenciada, pois é uma clara violação dos princípios e objetivos da UC. O incumprimento desta regra resulta na reprovação imediata à UC.

B Regras para Documentação de Programas Python

B.1 Introdução

Documentar programas em Python é essencial para garantir a compreensão e manutenção do código ao longo do tempo. Uma documentação clara e concisa não só ajuda o programador que escreveu o código, mas também ajuda outras pessoas que possam interagir com esse código no futuro. Em particular, permitirá aos professores compreender mais facilmente o seu código, seja para avaliação, seja para tirar dúvidas durante as aulas. Não seguir as regras de documentação pode resultar em dificuldades na compreensão do código, aumento da complexidade do processo de avaliação e, consequentemente, em descontos na nota de trabalhos ou exercícios.

Para documentar o código vamos utilizar *docstrings*[2] e *type hints*. Uma *docstring* é uma string literal que ocorre como a primeira instrução de uma função (ou outros elementos modulares do Python). Todas as funções não triviais devem ter *docstrings*. *Type hints* são indicações/sugestões do tipo do valor guardado nas variáveis. Todas as funções devem ter os seus parâmetros e valor de retorno devidamente documentados com *type hints*.

B.2 Docstrings

A documentação não deve descrever como o código está implementado, mas sim o que o código faz.

- Usar *docstrings* para documentar todas as funções:
  - Uma *docstring* deve ser incluída como a primeira instrução de uma função.
  - Utilize sempre aspas triplas para iniciar uma *docstrings* (mesmo em *docstrings* de uma única linha), pois facilita futuras expansões.
  - A *docstring* deve ser uma frase que termina com um ponto, descrevendo o propósito da função como um comando ("Faz isto.", "Retorna aquilo.").
  - Para funções mais complexas, introduza uma *docstrings* de múltiplas linhas, que inclua uma linha de resumo seguida por uma descrição mais detalhada.
- Conteúdo da *docstring*:
  - A *docstring* deve resumir o comportamento da função e documentar seus parâmetros, valor(es) de retorno, efeitos colaterais, exceções lançadas e restrições sobre quando pode ser chamada.
  - Parâmetros opcionais devem ser claramente indicados na *docstring*, juntamente com a explicação de seus papéis e as variáveis correspondentes.
  - Não deve usar linhas para lá do comprimento da página e nunca maiores que 80 caracteres. Sempre que necessário deve introduzir fins de linha por forma a garantir que não tem linhas maiores que a página.
- Comentários:
  - O código Python deve ser claro e autoexplicativo para quem o lê.
  - Cada linha de código deve ser simples de interpretar, evitando soluções excessivamente complexas sem justificação.
  - Os comentários devem ser utilizados apenas em situações especiais onde não é possível ter o código claro e autoexplicativo.

[2]Segue-se a nomenclatura usada nos PEP- Python Enhancement Proposals, nomeadamente PEP 8, PEP 257, PEP 484, PEP 483 e PEP 3107.

  - Excesso de comentários é considerado má prática e pode resultar num penalização na classificação.

- Todos os identificadores devem ser descritivos do seu significado:
  - Os nomes de variáveis descrevem o seu conteúdo de forma adequada.
  - Os nomes das funções indicam o que a função calcula.
  - Os nomes dos parâmetros indicam o que o parâmetro vai representar.
  - Expressões matemáticas complicadas devem ser particionadas em sub-expressões mais simples, utilizando variáveis auxiliares com nome adequado ao conteúdo, e só depois calcular o valor final com base nos valores parciais.

- Deverá ser possível testar automaticamente o código desenvolvido:
  - Para as principais funções, devem ser apresentadas funções de teste.
  - As funções de teste devem utilizar a instrução assert, conforme apresentado nas aulas teóricas.
  - Os testes utilizados devem ser o mais variados possível e endereçar casos gerais, casos extremos e condições de fronteira.

B.3 Type-Hints

Normalmente os *Type-Hints* são vistos como documentação extra do código e são fundamentais para garantir a clareza e a robustez do código Python. A sua presença facilita a compreensão do código e ajuda a detecter erros de tipo durante o desenvolvimento.

O código também poderá ser validado automaticamente com ferramentas apropriadas para validar os Type-Hints, por exemplo, as ferramentas `mypy` e `pylint`.

Assim, a utilização de *Type-Hints* é obrigatória em ICE e a sua ausência resulta em descontos na classificação.

B.3.1 Sintaxe

As anotações *Type-Hints* para parâmetros têm a forma de expressões (`int`, `float`, `str`, `bool` e *tipos compostos*) que seguem o nome do parâmetro. O retorno de valores segue-se aos parâmetros da função. Uma função foo que recebe dois parâmetros, um do tipo `float` e outro do tipo `int`, e que retorna um valor do tipo `str`, declara-se assim:

```
def foo(a: float, b: int) -> str:
    """Retorna a^b por extenso (texto)."""
```

```
    ...
    return ...
```

Para além de `float`, pode ser utilizado qualquer outra expressão válida como *Type Hint*. Deve ser utilizado o tipo **`None`** para anotar funções que não retornam nenhum valor.

```
def bar(a: float, b: int) -> None:
    """Função que não devolve nada."""
    ...
```

B.3.2 Tipos Compostos: Listas, Tuplos e Dicionários

Lista: representada como `list[tipo_elemento]`.

Tuplo: indicado listando os tipos de elementos, por exemplo, `tuple[int, int, str]`.

Dicionário: utilizado como `dict[tipo_chave, tipo_valor]`.

Observação: Listas e tuplos foram apresentados na aula teórica 3. Os dicionários foram apresentados na aula teórica 8.

B.4 Exemplos

Os seguintes exemplos serão utilizados para ilustrar a utilização concreta das regras de documentação apresentadas e são baseados em material apresentado nas aulas de ICE. Os conceitos de *docstrings* e *type hints* serão abordados, com indicações para a sua aplicação adequada. Igualmente, durante as aulas, é realçada a importância da documentação clara e concisa para facilitar o entendimento e manutenção do código Python. Lembre-se de documentar adequadamente o seu código e evitar o excesso de comentários, promovendo assim a escrita de código Python mais legível e compreensível.

```
def quadrado(x: float) -> float:
    """Calcula o quadrado do número x."""
    return x**2
```

Esta função ilustra a documentação de uma função simples recebendo apenas um parâmetro `float` e retornando igualmente um `float`. Repare que apesar de a documentação ter apenas uma linha, refere-se a variável parâmetro (`x`) e o que a função retorna. Igualmente, os *Type-Hints* indicam o domínio e contradomínio da função. Neste caso não é necessária mais nenhuma informação.

Outra função (apresentada nas aulas de forma mais sucinta, mas igualmente correta), irá calcular o comprimento de um segmento de reta. Eis uma forma possível e geral para apresentar esta função:

```
def comprimento_segmento(x1: float, y1: float,
                         x2: float, y2: float) -> float:
```

```
    """
    Calcula o comprimento do segmento de reta.
    O segmento é definido pelos pontos (x1, y1) e (x2, y2).
    Parâmetros:
    x1, y1: A coordenada x e y do primeiro ponto.
    x2, y2: A coordenada x e y do segundo ponto.
    Retorna: O comprimento do segmento de reta.
    """
    y = ((x2 - x1)**2 + (y2 - y1)**2)**0.5
    return y
```

Neste caso, começa-se por salientar a utilização de uma *docstring* com várias linhas. De facto, a descrição de todos os parâmetros não caberia numa única linha. Assim, tem de se seguir um formato mais standard. Como referido, indica-se o resumo numa única linha, seguido de descrição relevante. No fim especificam-se todos os parâmetros e o retorno da função. O tipo de dados (`float`) não necessita ser referido na *docstring*, pois já está no *type-hint*. Na assinatura da função, após a vírgula do segundo parâmetro (`x2`) e na linha de descrição, verificou-se que a linha estava muito longa e era conveniente quebrá-la.