

Módulo 03

Revisão de 10-11-2024

Programação Estruturada

Em algumas linguagens de programação existe uma diferença entre procedimentos e funções, em Python podemos utilizar os dois termos pois têm o mesmo significado.

1. Introdução às Funções

O que são funções - uma função é um bloco de código que só é executado quando chamado.

```
In [ ]: print("Olá")  
  
#definir a função  
def MinhaFuncao():  
    print("World")  
  
print("Mundo")
```

O código da função não é executado porque nunca foi **chamada**. Para chamar ou executar uma função utilizamos o nome da função com () à frente.

```
In [ ]: print("Olá")  
  
#definir a função  
def MinhaFuncao():  
    print("Hello World")  
  
print("Mundo")  
#chamar a função  
MinhaFuncao()
```

Execução do código passo a passo

Quando a função é chamada a execução do código **salta** para dentro da função até que todo o código da função é executado, de seguida a execução continua na linha seguinte onde a chamada da função tinha sido efetuada.

```
In [ ]: print("Antes da função")  
  
def FuncaoA():  
    print("Dentro da função")  
  
#chama a função  
FuncaoA()
```

```
print("Depois da função")
```

Enquanto o código da função não terminar de ser executado o resto do programa fica à espera. Quando uma função é executada é adicionada uma frame na call stack e são criadas as variáveis locais que possam existir dentro da função.

Porquê utilizar funções?

1. Reutilizar código: evitar repetir código
2. Organização: criar blocos de código facilita a sua leitura e compreensão
3. Abstração: funções fazem o que têm de fazer e não temos de saber os detalhes todos
4. Manutenção: se é necessário alterar alguma coisa, podemos simplesmente alterar o código de uma função sem ter de alterar o resto do código

Boas práticas na criação de funções em programação são essenciais para garantir a clareza, manutenibilidade e eficiência do código.

1. **Nome Claro e Descritivo:** O nome de uma função deve refletir claramente o que ela faz. Evite nomes genéricos e dê preferência a nomes que descrevam a ação realizada ou o resultado retornado.
2. **Funções Devem Ser Curtas e Focadas:** Uma função deve realizar uma única tarefa ou um grupo muito relacionado de tarefas. Se uma função está se tornando longa ou complexa, considere dividí-la em funções menores.
3. **Parâmetros Limitados:** Tente limitar o número de parâmetros de uma função. Muitos parâmetros podem tornar a função difícil de entender e usar.
4. **Evitar Efeitos Colaterais:** Uma boa função deve ser pura, ou seja, não deve ter efeitos colaterais (como alterar variáveis globais ou externas). Funções puras são mais fáceis de testar e depurar.
5. **Documentação:** Comente seu código e escreva uma documentação clara para suas funções. Isso deve incluir a finalidade da função, descrição dos parâmetros, o valor de retorno e quaisquer exceções ou casos especiais.
6. **Consistência no Estilo de Codificação:** Siga um estilo de codificação consistente. Isso inclui convenções de nomenclatura, formatação e estruturação do código.
7. **Use Tipos de Retorno e Parâmetros Explícitos:** Em linguagens que suportam tipagem, especifique os tipos de parâmetros e retorno. Isso melhora a legibilidade e reduz a probabilidade de erros.
8. **Tratamento de Erros:** Certifique-se de que a função lida adequadamente com situações de erro ou entradas inválidas, possivelmente lançando exceções ou retornando valores de erro.

9. **Evitar Dependências Globais:** Uma função deve, na medida do possível, evitar depender de variáveis ou recursos globais, para facilitar sua reutilização e teste.
10. **Testabilidade:** Escreva funções de forma que sejam fáceis de testar. Isso geralmente significa que elas não devem ter dependências ocultas e devem ter resultados previsíveis.

Documentar funções

A documentação de funções em Python é geralmente feita usando docstrings. Docstrings são strings literais que aparecem como a primeira declaração em uma função, método, classe ou módulo. Elas são usadas para explicar o propósito e o funcionamento do código. Aqui estão algumas diretrizes sobre como escrever boas docstrings em Python:

1. **Use Três Aspas Duplas:** Inicie e termine uma docstring com três aspas duplas (`"""`). Isso permite que a string se estenda por várias linhas.
2. **Descrição Breve e Clara:** A primeira linha da docstring deve ser uma descrição concisa do propósito da função. Esta linha deve ser uma frase completa terminando com um ponto.
3. **Detalhamento Opcional:** Após a linha de descrição inicial, você pode incluir um parágrafo mais detalhado explicando o comportamento da função, seus argumentos, seu valor de retorno e qualquer outra informação relevante.
4. **Parâmetros e Tipos:** Liste cada parâmetro e descreva brevemente seu propósito. Se o tipo do parâmetro não é óbvio, inclua isso também.
5. **Valor de Retorno:** Descreva o que a função retorna. Inclua o tipo de retorno se não for óbvio.
6. **Exceções:** Documente as exceções que a função pode levantar.
7. **Exemplos de Uso:** Se possível, inclua um ou mais exemplos de como usar a função. Isso pode ser particularmente útil para funções complexas ou com muitos parâmetros.
8. **Consistência:** Mantenha um estilo consistente em todas as suas docstrings. Se estiver trabalhando em um projeto em equipe, siga as convenções de documentação do projeto.
9. **PEP 257:** Para orientações mais detalhadas, consulte a PEP 257, que fornece convenções para **docstrings** em Python.

Exemplo:

```
In [ ]: def add(a, b):  
    """  
        Soma dois números e retorna o resultado.  
    """
```

```
Parâmetros:  
a (int): o primeiro número a ser somado.  
b (int): o segundo número a ser somado.  
  
Retorna:  
int: a soma de a e b.  
  
Exemplo:  
>>> add(2, 3)  
5  
"""  
return a + b
```

Este é um formato básico, e pode ser adaptado conforme necessário para suas funções específicas. A documentação clara e consistente é uma parte vital do desenvolvimento de software e ajuda a manter o código acessível e sustentável.

2. Definindo e chamando funções

Para definir uma função utilizamos a palavra reservada **def** seguida do nome da função e :

Por exemplo: def MostrarMensagem():

O nome das funções seguem as mesmas regras que o nome das variáveis: não podem começar por um número e não podem ter espaços em branco. Cuidado com as palavras reservadas ou nomes de funções repetidos. O nome da função inclui sempre () - parentesis

```
In [ ]: def SomaDoisNumeros():  
    x = 10  
    y = 20  
    resultado = x + y  
    print(f"A soma é {resultado}")
```

Para **executar** a função utilizamos o nome seguido dos parentesis:

```
In [ ]: SomaDoisNumeros()
```

Os () - parentesis - indica que a função deve ser executada, sem os parentesis estamos a referir o objeto da função sem que esta seja executada, ou seja, é só uma referência para a função.

```
In [ ]: SomaDoisNumeros
```

Para ser possível executar uma função esta tem de ser definida primeiro, se o computador ainda não encontrou a definição da função esta não pode ser chamada

```
In [ ]: funcaoA() #erro porque a função ainda não foi definida
```

```
def funcao():
    print("Dentro da função A")
```

Função main

Apesar de não ser obrigatório é uma boa prática definir uma função **main** no ficheiro que tem o código principal do programa. Esta função será o ponto de entrada onde o código começa a ser executado. Desta forma também podemos utilizar este ficheiro como um módulo que pode ser importado para outros projetos sem que o código seja executado inadvertidamente, o que acontece se importarmos um módulo que tem código fora das funções.

```
In [ ]: def main():
    # Código principal do programa
    pass

#só é executada a função main se a execução do programa começou por este ficheiro
if __name__ == "__main__":
    main()
```

3. Parâmetros e argumentos

A função **SomaDoisNumeros()** que definimos anteriormente não é muito útil. Porquê?

Para ser mais útil ela devia permitir somar quaisquer dois números e não somente 10 com o 20.

Para tornar a função mais útil vamos definir dois parâmetros para podermos "dar" à função os números que pretendemos somar.

```
In [ ]: def SomaDoisNumeros(x,y):
    """Função que soma dois números e mostra o resultado
    Keyword arguments:
    argument -- os dois números a somar
    Return: nada
    """
    #repara que os valores a somar não estão definidos dentro da função
    resultado = x + y
    print(f"A soma é {resultado}")
```

Esta versão da função é diferente da anterior uma vez que ela **não conhece** quais são os valores que vai somar. Agora quando chamamos a função temos de passar os argumentos, ou seja, os valores para os parâmetros definidos.

```
In [ ]: SomaDoisNumeros(10,20)
```

```
In [ ]: SomaDoisNumeros(5,12)
```

Agora a função é muito mais útil porque permite somar quaisquer dois números.

Os **parâmetros** são as variáveis que **só vão existir dentro da função** e cujos valores são definidos no momento em que a função é chamada.

Os **argumentos** são os valores que atribuimos aos parâmetros no momento da execução da função.

```
In [ ]: def MostraMensagem(mensagem):
         print(mensagem)

MostraMensagem("Olá mundo")
# esta linha dá erro porque a variável mensagem é um parâmetro da função MostraM
# e só existe dentro da função
print(mensagem)
```

Este conceito é MUITO IMPORTANTE: todos os parâmetros de uma função, bem como as variáveis que são definidas dentro de uma função, têm uma utilização (scope) limitado ao corpo da função. Isto significa que estas variáveis são criadas quando a função é chamada e são destruídas quando a execução da função termina.

Considerando o seguinte código:

```
In [ ]: def SubtraiDoisValores(x,y):
         resultado = x - y
         print(resultado)

SubtraiDoisValores(10,5)
```

A função **SubtraiDoisValores** tem dois parâmetros: **x** e **y** e uma variável local: **resultado**

Qualquer um destes parâmetros e/ou variáveis estão limitados ao corpo da função. Isto quer dizer que NÃO EXISTEM FORA DA FUNÇÃO.

```
In [ ]: def SubtraiDoisValores(x,y):
         resultado = x - y
         print(resultado)

SubtraiDoisValores(10,5)
# estas linhas dão erros porque as variáveis
# x,y e resultado só existem dentro da função SubtraiDoisValores
# todas elas são locais
print(x)
print(y)
print(resultado)
```

A única diferença entre a variável resultado e as variáveis x e y (que são parâmetros) é que estes últimos têm um valor atribuído quando a função é chamada.

Os argumentos podem ser valores constantes ou variáveis.

```
In [ ]: def SubtraiDoisValores(x,y):
         resultado = x - y
         print(resultado)

a = 10
```

```
b = 5
SubtraiDoisValores(a,b)
```

Assim podemos colocar a questão: o que acontece à variável que é utilizada num argumento se o valor do parâmetro for alterado dentro da função?

```
In [ ]: def SubtraiDoisValores(x,y):
    resultado = x - y
    print(resultado)
    x = 0
    y = 0

a = 10
b = 5
SubtraiDoisValores(a,b)
print(a,b)
```

E a resposta é: NADA.

A razão deve-se ao facto de em Python existirem **tipos de dados imutáveis**, como por exemplo: inteiros, strings e outros. Isto quer dizer que sempre que atribuirmos um valor novo a uma destes variáveis é criada uma variável nova com o novo valor. Assim, no exemplo, a variável **a** continua com o valor 10 e ao atribuir 0 ao parâmetro **x** na realidade é criada uma nova variável **x** e não é alterado o valor original do **a**

Para demonstrar este conceito podemos utilizar a função **id()** que mostra o endereço de memória de uma variável. Quando alterarmos o valor da variável o endereço muda porque é criada uma variável nova.

```
In [ ]: x = 10
print(id(x))
print(id(x))
x=15
print(id(x))
```

Mas e se pretendermos **passar** um valor que é calculado dentro da função para o resto do programa? Uma solução passa por fazer com que a função **devolva** um valor.

```
In [ ]: def Troco(pagar,dinheiro):
    dinheiro = dinheiro-pagar #esta Linha não altera o valor da variável carteira
    print(f"Pagou {pagar} e ficou com {dinheiro}")

carteira =100
Troco(50,carteira)
print(f"Tem na carteira {carteira}")
```

Com a instrução **return** a função **devolve ou retorna** um valor.

```
In [ ]: def Troco(pagar,dinheiro):
    dinheiro = dinheiro - pagar
    print(f"Pagou {pagar} e ficou com {dinheiro}")
    #vamos devolver o valor
    return dinheiro
```

```

carteira =100
#Vamos chamar a função e guardar o valor que retorna
carteira=Troco(50,carteira)
print(f"Tem na carteira {carteira}")

```

Opcionalmente podemos indicar o **tipo de dados que a função vai devolver** na sua definição.

```

In [ ]: def Troco(pagar,dinheiro) -> int:
    dinheiro = dinheiro - pagar
    print(f"Pagou {pagar} e ficou com {dinheiro}")
    #vamos devolver o valor
    return dinheiro

carteira =100
#Vamos chamar a função e guardar o valor que retorna
carteira=Troco(50,carteira)
print(f"Tem na carteira {carteira}")

```

Os tipos de dados podem ser int, str, float, bool, Any, entre outros que vamos estudar mais tarde como list, dict, set e tuple

A instrução **return** também permite terminar a execução do código da função, à semelhança da instrução break dentro de um ciclo.

```

In [ ]: def Troco(pagar,dinheiro):
    if dinheiro==0:
        print("Não tem dinheiro para pagar")
        return None
    dinheiro = dinheiro - pagar
    print(f"Pagou {pagar} e ficou com {dinheiro}")
    return dinheiro

carteira = 100
#Vamos chamar a função sem guardar o valor que retorna
Troco(50,carteira)
Troco(50,0)

```

Neste exemplo a função é executada sem guardar o valor de retorno.

Valores padrão para os parâmetros

É possível definir um valor padrão para um parâmetro, esse valor só será utilizado caso a função seja chamada sem um argumento para esse parâmetro.

```

In [ ]: def Saudacao(texto="mundo"):
    print(f"Olá, {texto}")

Saudacao() #função chamada sem argumentos por isso texto assume o valor padrão
Saudacao("Joaquim") #neste caso o valor padrão não é utilizado

```

Neste exemplo o parâmetro texto tem um valor padrão que é utilizado da primeira vez que a função é chamada porque nessa chamada não é fornecido nenhum valor, ou seja, a função foi chamada sem argumentos. Da segunda vez a função é chamada com o

argumento "Joaquim" sendo esse o valor utilizado pelo parâmetro ao invés do valor padrão.

NB: Quando definimos um valor padrão para um parâmetro temos de definir para todos os parâmetros que se encontram à sua direita, uma vez que o computador não conseguiria saber se o valor que era indicado era para o primeiro ou para o segundo parâmetro

```
In [ ]: # se definimos um valor padrão para o parâmetro x, este deve ser o parâmetro maior ou temos de definir para todos os parâmetros que se encontram à sua direita
def Soma(x=0,y):
    resultado = x + y
    print(resultado)
```

Nomear os parâmetros

Ao chamar a função os argumentos são atribuídos aos parâmetros pela ordem que foram definidos, ou seja, da esquerda para a direita, no entanto podemos tornar o nosso código mais legível se indicarmos o nome dos parâmetros e assim até podemos trocar a ordem dos parâmetros.

```
In [ ]: def Subtrai(x,y):
    resultado = x - y
    print(resultado)

Subtrai(10,5) #chamada da função com os valores 5 e 10 que são atribuidos aos parâmetros x e y respectivamente
Subtrai(y=5,x=10) #chamada da função nomeando os parâmetros e invertendo a ordem
Subtrai(5,10)
```

Variáveis Globais

Como já foi dito todos os parâmetros e variáveis declaradas dentro das funções são consideradas variáveis locais, ou seja, a sua utilização (scope) está limitada ao corpo da função.

```
In [ ]: def MostraValor(numero):
    print(numero)    # a variável numero só existe dentro da função

MostraValor(10)
print(f"Valor do número: {numero}") #esta Linha dá erro porque a variável numero
```

No entanto podemos definir **variáveis globais** que é possível utilizar em qualquer parte do nosso código, dentro e fora das funções.

```
In [ ]: numero = 10 #esta variável é global assim é possível utilizá-la em qualquer parte do código
def SomaDoisNumeros():
    print(numero) # não dá erro porque a variável é global

MostraValor()
print(numero)    #variável é global
```

Assim podemos cair na tentação de definir as nossas variáveis todas sempre como globais e não temos o problema de ter de controlar se a variável existe ou não em alguma parte do nosso código, mas as variáveis globais podem criar outros problemas, como se pode ver pelo código que se segue:

```
In [ ]: resultado = 0 #esta variável é global assim é possível utiliza-la em qualquer pa  
  
def SomaDoisNumeros(x,y):  
    resultado = x + y    #estamos a utilizar a variável global ou a criar uma nov  
    print(resultado) # não dá erro porque a variável é global  
  
SomaDoisNumeros(10,5)  
print(resultado)  #variável é global
```

Mas depois temos problemas como este:

```
In [ ]: #variáveis globais  
resultado = 0  
x=10  
y=5  
  
def SomaDoisNumeros():  
    resultado = x + y    #estamos a utilizar as variáveis globais?  
    print(resultado) # não dá erro porque a variável é global  
    x=0 #esta linha provoca um erro porque x passa a ser local! (obrigado Python)  
  
SomaDoisNumeros()  
print(resultado)
```

O erro ocorre porque o Python deteta que estamos a alterar o valor da variável X e então cria uma variável local com o mesmo nome da variável global, se o que pretendemos é mesmo alterar a variável global temos de alterar o código:

```
In [ ]: soma = 0 #esta variável é global assim é possível utiliza-la em qualquer parte d  
  
def SomaDoisNumeros(x,y):  
    global soma # assim indicamos que a variável soma que vamos utilizar dentro  
    soma = x + y    #estamos a utilizar a variável global  
    print(soma) # não dá erro porque a variável é global  
  
SomaDoisNumeros(5,10)  
print(soma)  #variável é global
```

A instrução **global** indica que a variável que vai utilizada é a variável global e assim não é criada uma variável local dentro da função.

A utilização de variáveis globais deve ser limitada pois cria código mais complicado de perceber e de manter!

Call Stack

Nesta parte vamos tentar perceber, com maior detalhe, como é que as funções são executadas pelo computador.

Como já vimos ao chamar uma função a execução salta para dentro da função e o código desta é executado. E este princípio aplicação da mesma forma se dentro da função for chamada outra função.

```
In [ ]: variavel_global=10

def FuncaoA():
    variavel_local_funcao_a=5
    print("Esta é a função A")

def FuncaoB():
    variavel_local_funcao_b=15
    print("Inicio da função B")
    FuncaoA()
    print("Fim da função B")

print("No programa principal")
FuncaoA()
print("No programa principal")
FuncaoB()
print("No programa principal")
```

Sempre que uma função é chamada é criado um espaço de memória denominado de frame. É neste frame de memória que são criadas as variáveis locais. Ao concluir a execução da função esta frame é apagada e toda a informação que tinha perde-se.

Cada frame é criada e gerida pelo computador como se de uma pilha (stack) de pratos de trata-se, em que o último prato colocado na pilha é sempre o primeiro a ser retirado, é por isso que, no código exemplo, ao chamar a FuncaoA dentro da FuncaoB estamos a colocar nessa pilha

[Demonstração da call stack](#)

Módulos

Agora que já sabemos criar funções podemos criar os nossos próprios módulos de funções e importar para os nossos programas e assim maximizar a reutilização do código.

Vamos criar um módulo com duas funções

```
In [ ]: %%writefile meu_modulo.py

def saudacao(nome):
    return f"Olá, {nome}!"

def soma(a, b):
    return a + b
```

Agora para utilizar este módulo temos de o importar

```
In [ ]: import meu_modulo #importação do módulo

resultado = meu_modulo.soma(10, 5)
print(resultado) # Saída: 15

mensagem = meu_modulo.saudacao("Maria")
print(mensagem) # Saída: Olá, Maria!
```

Instalar módulos antes de importar

Alguns módulos são disponibilizados em repositórios e têm de ser instalados no nosso computador antes de serem importados para o programa.

Para instalar módulos utiliza-se a ferramente *pip*

Por exemplo, um módulo que pode ser útil é o `termcolor` que permite a utilização de texto com cores no terminal.

Para instalar o módulo no computador executamos o comando: **pip install termcolor**

Depois já podemos importar:

```
In [ ]: from termcolor import colored

def imprimir_texto_cor(texto, cor):
    print(colored(texto, cor))

imprimir_texto_cor("Olá, mundo!", "green")
imprimir_texto_cor("Olá, mundo!", "red")
```

Recursividade

Uma função recursiva é aquela que se chama a si mesma para resolver uma versão menor do problema. Na prática é um ciclo sem utilizar as instruções **for** ou **while**. Ao chamar a função dentro do seu próprio código estamos a promover a repetição da sua execução.

Um aspeto fundamental do código de uma função recursiva é a condição para parar de se chamar a si própria. É o caso base. Sem esta condição criamos um ciclo infinito de chamadas à função que resulta num erro de Stack Overflow, em que esgotamos o limite de chamadas máximas da stack de execução do programa.

Exemplo: Uma função que calcula a exponenciação. Os parâmetros são o valor da base e do expoente. Como condição de paragem podemos definir o caso base em que o expoente é 1 que, como de certo sabe, resulta no valor da base, ou seja, qualquer valor elevado a 1 dá o próprio valor.

Assim para resolver 2^4 podemos dizer que é igual a $2 * 2^3$

Por sua vez 2^3 é $2 * 2^2$ e assim sucessivamente.

```
In [ ]: def Expoente(base,expoente):
    #Condição de paragem
    if expoente==1:
        return base
    else:
        #recursividade: a função chama-se a si própria
        return base*Expoente(base,expoente-1)

print(Expoente(2,4))
```

Funções com yield

As funções são blocos de código em que as variáveis são destruídas quando a execução da função termina, por isso não guardam o estado entre chamadas da mesma função.

No entanto podemos colocar a execução de uma função num estado de pausa, em que a execução é interrompida até que seja novamente chamada.

```
In [ ]: def gerar_pares(limite):
    n = 0
    while n <= limite:
        yield n      # interrompe a execução e devolve o valor de n
        #na próxima vez que a função é chamada continua na Linha seguinte
        #à instrução yield
        n += 2

    # Usando o gerador
    for numero_par in gerar_pares(10):
        print(numero_par)
```

Módulos do Python

O Python tem um grande número de módulos que podem ser importados para os nossos programas. Alguns destes já fazem parte da estrutura base da linguagem enquanto que outros têm de ser instalados utilizando a ferramenta pip.

Módulo SYS

O módulo **sys** é um dos módulos incorporados (não requer instalação) e que fornece acesso a algumas funções e variáveis que dizem respeito ao interpretador do Python.

1. **sys.argv**: Lista de strings que contém os argumentos da linha de comando passados para um script Python. **arg[0]** é o nome do script.

2. **sys.exit()**: Permite interromper a execução de um script Python e terminar a sua execução retornando ao ambiente da linha de comandos do shell. Opcionalmente pode ser indicado um código de interrupção.
3. **sys.path**: Lista de caminhos para as pastas onde o Python procura por módulos a serem importados.
4. **sys.stdin, sys.stdout, sys.stderr**: Objetos que correspondem à entrada e saída padrão e ainda à saída de erros do Python.
5. **sys.version**: String que indica a versão do interpretador Python.
6. **sys.platform**: String que indica a plataforma em que o Python está a ser executado.
7. **sys.modules**: Dicionário que relaciona os nomes dos módulos importados com os módulos em si. Permite verificar que módulos já foram importados.
8. **sys.executable**: Um string que representa o caminho para o executável do interpretador Python.

Módulo OS

O módulo **OS** do Python permite ter acesso a funcionalidades e funções do sistema operativo.

1. **os.name**: String com o nome do Sistema Operativo
2. **os.getcwd()**: String com o caminho para a pasta atual.
3. **os.chdir(caminho)**: Muda a pasta atual para o caminho indicado.
4. **os.listdir(caminho=''): Devolve a lista dos ficheiros e pastas do caminho indicado.
5. **os.mkdir(caminho, mode=0o777)**: Cria uma pasta com o caminho indicado
6. **os.makedirs(caminho, mode=0o777, exist_ok=False)**: Cria uma pasta recursivamente, incluindo as pastas necessárias.
7. ***os.rmdir(caminho)**: Apaga uma pasta vazia.

8. **os.removedirs(caminho)**: Apaga uma pasta recursivamente, incluindo as pastas intermediárias se vazias.

9. **os.remove(caminho)** ou **os.unlink(caminho)**: Remove um ficheiro.

10. **os.rename(origem,destino)**: Altera o nome de um ficheiro.

11. **os.stat(caminho)**: Obtém informações sobre o ficheiro ou pasta indicados (tamanho, data de modificação, etc)

12. **os.environ**: Dicionário com as variáveis de ambiente.

13. **os.system(comando)**: Executa um comando na shell do sistema operativo.

14. **os.path**: Módulo dentro do módulo OS que fornece funções para manipulação de caminhos de ficheiros, como **os.path.join**, **os.path.split**, **os.path.exists**, **os.path.isdir**, **os.path.isfile**, etc

Módulo math

1. **Funções de Trigonometria**: **sin(x)**, **cos(x)**, **tan(x)**, **asin(x)**, **acos(x)**, **atan(x)**

2. **Conversão de ângulos**: **radians(x)** e **degrees(x)**

3. **Funções hiperbólicas**: **sinh(x)**, **cosh(x)**, **tanh(x)**, **asinh(x)**, **acosh(x)**, **atanh(x)**

4. **Exponenciação e logaritmos**: **exp(x)**, **log(x[,base])**, **log2(x)**, **log10(x)**

5. **Potências e raiz quadrada**: **pow(x,y)**, **sqrt(x)**

6. **Constantes matemáticas**: **pi**, **e**, **tau** (2pi), **inf** (infinito) **nan** (Not A Number)

7. **Arredondamento e absoluto**: **ceil(x)** arredonda para cima, **floor(x)** arredonda para baixo e **fabs(x)** valor absoluto

8. **Funções especiais**: **factorial(x)** para calcular o factorial de um nº, **gdc(x,y)** para o máximo divisor comum, **isclose(a,b)** para verificar se dois valores são aproximadamente iguais.

Módulo datetime

Este módulo permite manipular datas e horários.

1. Classes de datas e horários:

- **datetime.date**: Representa uma data (ano, mês, dia)
- **datetime.time**: Representa um horário (hora, minuto, segundo, microsegundo)
- **datetime.datetime**: Combina uma data e um horário
- **datetime.timedelta**: Representa uma duração, ou seja, a diferença entre duas datas ou dois tempos.

2. Funções Úteis:

- **datetime.now()**: Data e hora atual
- **datetime.today()**: Data atual
- **datetime.combine(date, time)**: Cria um objeto datetime combinando um objeto date e um time
- **datetime.strptime(date_string, format)**: Converte uma string de uma data/hora para um objeto datetime

3. **Formatar e analisar datas**: As funções **strftime(formato)** e **strptime(data_string, formato)** são utilizadas para formatar datas. **strftime** converte um objeto date/hora numa string enquanto que a função **strptime** converte uma string num objeto.

4. **Atributos e métodos das classes**: Os objetos date e time têm várias propriedades como **year**, **month**, **day**, **hour**, **minute**, **second** e **wekkday()**

Módulo Random

Este módulo permite gerar números pseudoaleatórios.

1. **random.seed(a=None, version=2)**: inicializa o gerador de números aleatórios.

2. **random.random()**: Gera um número aleatório entre 0,0 e 1,0

3. **random.uniform(a,b)**: Gera um número aleatório entre a e b (inclusive)

4. **random.randint(a,b)**: Gera um número inteiro entre a e b (inclusive)