Aula 3 - PARI

Sumário

```
Introdução ao git e ao github
Tipos de dados (continuação)
Tempo e temporizadores em python
Tuples, Named tuples and classes
```

Controlo de versões

Crie um repositório no github para esta aula. Para isso terá de criar uma conta no github. A sua conta pode ter vários repositórios com vários projetos. Um exemplo:

https://github.com/miguelriemoliveira

Depois deve criar um repositório chamado pari_2020 e descarregá-lo para o seu computador com o comando git clone:

```
git clone <url do repositório>
```

Exercício 1 - Tempo e Temporizadores

A utilização de estruturas de tempo em python está bastante facilitada tutorial

O comando

```
from time import time
time()
```

devolve o número de segundos que passaram desde o início de uma época (tipicamente 1 de janeiro de 1970).

para obter a data:

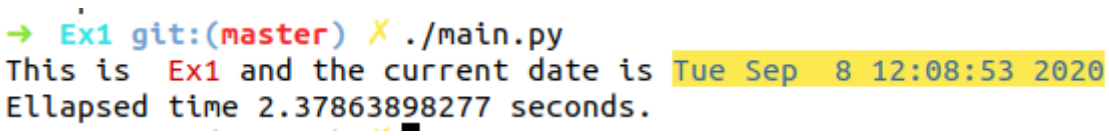
```
from time import time, ctime
ctime()
```

A utilização do primeiro formato permite realizar a subtração direta de modo a obter a duração em segundos entre dois tempos. Assim, pretende-se desenvolver um main.py que contenha duas funções tic() e toc() à semelhança do matlab.

Escreva um programa que realize a medição do tempo que demorou a executar um ciclo que

calcula a raiz quadrada dos números de 0 até 50 milhões. O programa deve imprimir a data atual ao iniciar e depois uma mensagem com o tempo que demorou a calcular as raízes quadradas.

Usando o package colorama, implemente mensagens coloridas com a informação destacada, por exemplo:



Exercício 2 - Números complexos

O objetivo é implementar código python que faça a gestão de operações com números complexos. Uma revisão rápida sobre números complexos aqui.

Numa primeira fase pretende-se criar funções isoladas para adicionar, multiplicar e imprimir números complexos:

Vamos utiizar um *tuple* para definir números complexos, em que o tamanho do tuplo é dois, o primeiro item contem a parte real e o segundo item contem a parte imaginária. Por exemplo, o número `5 + 3i` deverá ser representado pelo tuplo `(5,3)`

Criar as seguintes funções num script main.py:

```
addComplex(x,y);
multiplyComplex(x,y);
printComplex(x);
```

Testar as funções a partir do main() no ficheiro principal como descrito adiante. `addComplex()` adiciona dois números complexos e `multiplyComplex()` multiplica dois números complexos. O resultado é um número complexo, e estas funções devem implementar as operações fundamentais para os cálculos. A função `printComplex(x)` deve imprimir no ecran o texto no formato `a+bi` onde `a` é a perte real e `b` a parte imaginária do número `x` passado como argumento. As duas primeiras funções retornam tuplos de números complexos, e a última não retorna nada.

```
def addComplex(x, y):
    # add code here ...

def multiplyComplex(x, y):
    # add code here ...

def printComplex(x):
    # add code here ...

def main():
    # ex2 a)

    # define two complex numbers as tuples of size two
    c1 = (5, 3)
    c2 = (-2, 7)

    # Test add
    c3 = addComplex(c1, c2)
    printComplex(c3)

    # test multiply
    printComplex(multiplyComplex(c1, c2))

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

Exercício 3 - *Named Tuples*

A convenção utilizada para definir números complexos no exercício anterior não é explícita, no sentido em que o programador tem de se lembrar de que o primeiro item do tuple é a parte real e o segundo é a parte imaginária. Neste contexto parece uma informação fácil de memorizar, mas com o aumento da complexidade dos programas cada necessidade de memorizar algo acrescenta mais uma "ponta solta".

Assim, é mais **pythonic** utilizar uma forma de representação da informação que seja realmente explícita. Sugere-se a utilização de *namedTuples* que são tuplos que permitem nomear cada um dos items para posterior utilização.

Para definir um *NamedTuple* pode usar o seguinte código:

```
from collections import namedtuple

Complex = namedtuple('Complex', ['r', 'i'])


def addComplex(x, y):
    # addapt code to use named tuples

def multiplyComplex(x, y):
    # addapt code to use named tuples

def printComplex(x):
    # addapt code to use named tuples

def main():
    # define two complex numbers as tuples of size two
    c1 = Complex(5, 3)  # use order when not naming
    c2 = Complex(i=7, r=-2)  # if items are names order is not relevant
    print('c1 = ' + str(c1)) # named tuple looks nice when printed

    # Test add
    addComplex(c1, c2)
    printComplex(c3)

    # test multiply
    printComplex(multiplyComplex(c1, c2))


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

Exercício 4 - Utilização de classes

A utilização de *namedtuples* pode ser vantajosa por comparação com muitas outras abordagens. No entanto, no código desenvolvido no exercício 3 há ainda um ponto pouco claro. As funções respeitantes aos números complexos estão isoladas. Claro, neste caso isto não é um grande problema visto que não há outras funções, mas imagine que tinha outras 50 funções dedicadas a outras tarefas. Quando assim é, importa organizar a informação e as funcionalidades em estruturas de dados que encapsulem um determinado assunto.

As classes são a componente principal da programação orientada a objetos. Ver informação sobre classes em python Neste caso pode usar-se uma classe que contenha não só a definição do número complexo mas também as funcionalidades para o operar e imprimir.

```
class Complex:

    def __init__(self, r, i):
        self.r = r  # store real part in class instance
        self.i = i  # store imaginary part in class instance

    def add(self, y):
        # addapt code to use classes

    def multiply(self, y):
        # addapt code to use classes

    def __str__(self):
        # addapt code to use classes

def main():
    # declare two instances of class two complex numbers as tuples of size two
    c1 = Complex(5, 3)  # use order when not naming
    c2 = Complex(i=7, r=-2)  # if items are names order is not relevant

    # Test add
    print(c1)  # uses the __str__ method in the class
    c1.add(c2)
    print(c1)  # uses the __str__ method in the class

    # test multiply
    print(c2)  # uses the __str__ method in the class
    c2.add(c1)
    print(c2)  # uses the __str__ method in the class
```