Aula 1 – Python (para programadores de Java!)

Nesta aula vamos ver de forma sucinta como programar em Python. É um pressuposto que estão adquiridos todos os conceitos de programação, sendo o foco na utilização do Python como linguagem alternativa ao Java. É também um pressuposto o conhecimento de como utilizar os sistemas operativos disponíveis no DI, nomeadamente em relação ao acesso e à execução das aplicações disponíveis.

A versão do Python a adotar é a 3.7.

Recursos

Alguns recursos úteis que poderá sempre usar, e até já na aula de hoje.

- Sítio official - https://www.python.org - Para *downloads*, documentação completa, …
- Documentação principal oficial:
  - Tutorial de Python 3 - https://docs.python.org/3/tutorial/ -
  - Standard Library - https://docs.python.org/3/library/ -
  - Language Reference - https://docs.python.org/3/reference/ -
  - Style Guide - https://www.python.org/dev/peps/pep-0008/ - convenções a usar na escrita do código Python.
- A Wiki de Python - https://wiki.python.org/moin/
- Python Tutor - http://pythontutor.com/ - Execução online de programas com possibilidade de visualização da evolução do estado do programa ("live programming mode")
- Tutorialspoint – Documentação sintética do essencial para começar a programar em Python (https://www.tutorialspoint.com/python3/)
- Python Course - https://www.python-course.eu/python3_course.php -
- Livros online:
  - *Think Python: How to Think Like a Computer Scientist*, 2nd edition (PDF, HTML)
    - Este livro tem uma versão interactiva (https://runestone.academy/runestone/static/thinkcspy/index.html)
  - e muitos outros na Wiki de Python: https://wiki.python.org/moin/PythonBooks

Ambiente de desenvolvimento

Existem inúmeros ambientes de desenvolvimento que suportam Python. Nos laboratórios do DI têm dois à disposição:

- **IDLE** – fornecido com o pacote standard oficial. Leve, com algumas limitações.
- **Anaconda** – plataforma que inclui um IDE mais artilhado e muitas bibliotecas não standard do Python. Escolha, nesta plataforma, a ferramenta **spyder**.

Vamos assumir o IDLE como ambiente de desenvolvimento.

Ao abrir a respetiva aplicação, notem que se pretende usar o Python 3 (e não o Python 2, também disponível). Verificar, portanto, se a versão é a pretendida. Deverá aparecer, na janela "Python Shell", algo como:

```
Python 3.6.2 (v3.6.2:5fd33b5926, Jul 16 2017, 20:11:06)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
```

Python é uma linguagem interpretada. O *prompt* do interpretador é o símbolo '>>> '. Assim que este aparece, poderão ser dadas instruções que serão imediatamente executadas. Poderão também ser avaliadas expressões.

```
>>> x = 10
>>> print(x*2)
20
>>> x*4
40
```

A forma mais comum de trabalhar consiste em ter abertas duas janelas:

- Uma com a *Shell*, aberta automaticamente quando se arranca o IDLE (ilustrada no exemplo acima)
- Outra para editar o ficheiro que vai conter o programa (aberta no menu *File -> New File*)

O primeiro programa que sempre fazemos é o "Hello World!".
Em Python, é simplesmente um ficheiro com uma linha de código:

```
print("Hello World!")
```

Para executarmos um programa em Python temos que gravá-lo num ficheiro de extensão '.py' (por exemplo *hello.py*) e seguidamente 'Run -> Run Module' (ou simplesmente a tecla F5).
A sua execução será visível na janela do interpretador (*Shell*):

```
>>>
=== RESTART: /Users/..../hello.py ===
Hello
World!
>>>
```

A indicação "RESTART…" indica que o interpretador é inicializado (sendo eliminados todos os símbolos criados na sessão), o que sempre acontece cada vez que se executa um programa como indicado.

Compare-se este programa em Java e Python:

HelloWorld.java	hello.py

`public class HelloWorld`
`{  public static void main (String[] args)`
`  {`
`  System.out.println("Hello World!");`
`  }`
`}`
	`print("Hello World!")`

Este exemplo ilustrou uma das vantagens do Python: o ser uma linguagem que possibilita o desenvolvimento de programas muito mais concisos.

Conceitos básicos do Python

Podemos identificar duas características chave do Python (e que correspondem a diferenças importantes face ao Java):

- Tipos estáticos (Java) vs dinâmicos (Python) – Em Python, o tipo das variáveis é inferido a partir do contexto (em vez de declarado explicitamente), podendo ser alterado durante a execução do programa. Veja-se o exemplo:

```
>>> x = 10
>>> type(x)
<class 'int'>
>>> x = False
>>> type(x)
<class 'bool'>
>>> x = 10.0
>>> type(x)
<class 'float'>
>>> x = "Hello!"
```

A função *type()* permite saber o tipo de um objecto.

- Indentação obrigatória – Em Python, um bloco de instruções é identificado pela sua indentação (em Java é pela colocação de { }). Esta distinção será ilustrada nos próximos exemplos.

Outras diferenças:

- Comentários começam com o caracter '#'.
- Não se coloca nenhum caracter para terminar um comando (como o ';' em Java), pois a estrutura é determinada pela indentação.
- Não é necessário ter o código dentro de uma classe.
- O nome de um ficheiro é independente do nome de uma classe que represente.
- Num mesmo ficheiro podemos ter várias classes.

Tipos simples e compostos

No exemplo anterior estão ilustrados os três tipos simples existentes em Python: *int*, *float* e *bool*. Para manipular os valores destes tipos estão disponíveis os operadores habituais (veja a lista de todos eles em (por exemplo):
https://www.tutorialspoint.com/python/python_basic_operators.htm

Para além dos tipos simples, existem os seguintes tipos compostos:

- *str* – strings, por exemplo: `"Hello!", "111", ou 'World'`
- *list* – listas, por exemplo: `[1,2,3],['a','b','f'], ou [1,'w',7.5,False]`
- *tuple* – tuplos, por exemplo: `(1,2,3), ('a','b','f'), ou (1,'w',7.5,False)`

Estes são tipos sequenciais, sendo os seus elementos indexados a partir de 0. Exemplos:

```
>>> s = "Hello!"
>>> type(s)
<class 'str'>
>>> s[1]
'e'
>>> len(s)
6

>>> lista = [1, 'w', 7.5, False]
>>> lista[2]
7.5
```

A diferença principal entre listas e tuplos é que estes são imutáveis (não podem ser modificados, tal como as strings). Relativamente às listas, é possível alterar o valor dos seus elementos. Exemplos:

```
>>> lista
[1, 'w', 7.5, False]
>>> lista[3] = 8
>>> lista
[1, 'w', 7.5, 8]

>>> tuplo = (1, 'w', 7.5, False)
>>> type(tuplo)
<class 'tuple'>
>>> tuplo[3] = 8
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
tuplo[3] = 8
TypeError: 'tuple' object does not support item assignment
>>>
```

Para programar com os tipos sequenciais, existe um operador que facilita bastante a manipulação dos valores: o slice (‘[:]’), que permite extrair, de forma sintética, alguns elementos de uma sequência. Uma utilização simples corresponde a indicar os índices de forma a extrair uma subsequência. Por exemplo:

```
>>> s = "Hello World!"
>>> len(s)
12
>>> s[6:12]
'World!'
>>>
```

A operação de slicing ilustrada acima extrai da string a sub-sequência com os índices de 6 a 11. Mais exemplos e mais formas de usar por exemplo na secção “Slicing” em: https://www.python-course.eu/python3_sequential_data_types.php

Em Python os valores de uma sequência podem também ser acedidos usando índices negativos. O índice -1 representa o último elemento da lista; o -2 o penúltimo, e assim sucessivamente. Isto é válido para qualquer tipo de sequência (lista, tuplo, ou string):

```
>>> numeros = [3,7,5,2,6]
>>> numeros[-1]
6
>>> numeros[-5]
3
>>> numeros[-4:-1]
[7, 5, 2]
>>> numeros[-4:4]
[7, 5, 2]
>>> numeros[-1:-4]
[]
```

Existem ainda os dicionários (*dict*), estruturas tipo "mapa" que permitem guardar pares chave/valor, com acesso por chave.

```
>>> dic = {'a':10, 'b':20, 'x':40}
>>> type(dic)
<class 'dict'>
>>> dic['b']
20
>>> dic['c'] = 25
>>> dic
{'a': 10, 'b': 20, 'x': 40, 'c': 25}
```

Comandos principais

Para além da afetação, da qual já foram ilustrados vários exemplos, os comandos principais de qualquer linguagem imperativa são as que permitem concretizar estruturas condicionais e repetitivas.

Vejamos os três principais em Python (comparando com Java)

Java - if	**Python - if**

`if (cond) {`
`  a = a + 1;`
`  cond = false;`
`}`
`else{`
`  a = a - 1;`
`}`
	`if cond :`
	`  a = a + 1`
	`  cond = False`
	`else :`
	`  a = a - 1`

Notar a importância da indentação em Python, bem como a colocação dos ':' no fim das linhas do 'if' e do 'else'. O símbolo ':' indica que na linha seguinte se iniciará um novo bloco, que deverá vir indentado (e terá que ter todas as suas instruções com a mesma indentação).

Java - for	**Python - for**

`// Imprimir os inteiros de 1 a 10`
`for (int i = 1; i <= 10; i++)`
`{`
`  System.out.println(i);`
`}`
	`# Imprimir os inteiros de 1 a 10`
	`for i in range(1,11):`
	`  print(i)`

A função *range()* permite gerar sequências de inteiros num dado intervalo, com um dado passo. A sequência pode ser gerada passo a passo (como acontece no exemplo acima) ou convertida em lista, como nos exemplos seguintes:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
```

```
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
```

Java - while	**Python - while**

`s = "";`
`contador = 5`
`while (contador > 0) {`
`s = s + "O";`
`contador--;`
`}`
`mystring = s;`
	`s = ""`
	`contador = 5`
	`while contador > 0:`
	`    s = s + "O"`
	`    contador -= 1`
	`mystring = s`

Notar, na versão Python, a necessidade de indentação do bloco do ciclo while e a desnecessidade da colocação dos parenteses na guarda do ciclo.

Definição de funções

Funções são definidas recorrendo à palavra reservada 'def'.

```
def f(x) :
    s = 0
    for i in range(x) :
        s = s + i
    return s
```

Notar novamente os ':' na primeira linha, precedendo o bloco da função (que tem que ir corretamente indentado), neste exemplo uma sequência de três instruções, a última das quais (*return*) indicando o valor a devolver pela função.

Após definir a função poderemos invocá-la:

```
>>> f(5)
10
```

Input / Output

Já vimos em vários exemplos que o output é feito com a instrução print().
Para realizar input, utiliza-se a função *input()* que devolve sempre uma string com o input fornecido. A conversão para o tipo pretendido tem que ser feita explicitamente. Exemplo:

```
>>> x = input("Escreva um valor inteiro: ")
Escreva um valor inteiro: 22
>>> x
'22'
>>> y = int(x) // 2
>>> print("{1} corresponde a metade de {0}.".format(x,y))
11 corresponde a metade de 22.
```

Notar no exemplo acima o output formatado: na string de formatação, os pares de {} determinam os locais onde aparecerão os valores indicados em "format()", referindo os índices respetivos (no caso acima, x tem índice 0, pois é o primeiro argumento, e y o índice 1). Exemplos detalhados sobre a forma de formatar output na secção *"The Pythonic Way: The string method "format"* de https://www.python-course.eu/python3_formatted_output.php

Exercícios

1. Se listarmos todos os números naturais inferiores a 10 que são múltiplos de 3 ou 5, obtemos 3, 5, 6 e 9. A soma desses múltiplos é 23.

   Encontre a soma de todos os múltiplos de 3 ou 5 abaixo de 1000.

2. Cada novo termo na sequência Fibonacci é gerado pela adição dos dois termos anteriores. Ao começar por 1 e 2, os 10 primeiros termos serão:

   1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

   Ao considerar os termos na sequência de Fibonacci cujos valores não excedem quatro milhões, encontre a soma dos termos de valor par.

3. Escreva um programa que lê um número inteiro e calcula a soma dos seus dígitos. Verifique que o input é válido. Caso contrário, apresente a mensagem de erro "Input Inválido: introduza um numero inteiro".

4. Repita o exercício 3 definindo duas funções: *le_inteiro()* e *soma_digitos(inteiro)*.

5. Escreva a função *apaga_ocorrencias(tuplo, inteiro)* que recebe um tuplo de inteiros e um número inteiro e devolve um tuplo em que as ocorrências do inteiro foram apagadas. Faça duas versões da função, utilizando e não utilizando *range*.

6. Repita o exercício 5 usando listas. Faça duas versões da função *apaga_ocorrencias(lista, inteiro)* uma que não altera a lista inicial devolvendo uma nova lista e outra que altera a lista inicial.

Programação Orientada a Objectos

A programação orientada a objectos em Python dá muito mais liberdade ao programador e, mais uma vez, permite uma codificação muito mais concisa. Veja-se o seguinte exemplo comparativo:

Java -

```
public class Employee
{
    private String myEmployeeName;
    private int    myTaxDeductions = 1;
    private String myMaritalStatus = "single";


    //--------- constructor #1 -------------
    public Employee(String EmployeName)
    {
        this(employeeName, 1);
    }


    //--------- constructor #2 -------------
    public Employee(String EmployeName, int taxDeductions)
    {
       this(employeeName, taxDeductions, "single");
    }


    //--------- constructor #3 ------------
    public Employee(String EmployeName,
           int taxDeductions,
           String maritalStatus)
    {
       this.employeeName    = employeeName;
      this.taxDeductions   = taxDeductions;
     this.maritalStatus    =  maritalStatus;
}
...
```

Python -

```
class Employee():
     def __init__(self, employeeName, taxDeductions=1, maritalStatus="single"):
         self.employeeName    = employeeName
         self.taxDeductions   = taxDeductions
         self.maritalStatus   = maritalStatus
...
```

Em Python há apenas um construtor, o método *__init__()*, que, no entanto, pode ser invocado de formas alternativas. Por exemplo, assumindo a definição acima:

```
>>> x = Employee("João")
>>> y = Employee("Maria",2)
>>> z = Employee("Manuel",maritalStatus="married")


>>> for p in [x,y,z] :
     print("{0},{2},{1}".format(p.employeeName,p.taxDeductions,p.maritalStatus))


João,single,1
Maria,single,2
Manuel,married,1
```

Algumas observações:

- O Python admite a definição de valores por omissão para os argumentos das funções (e métodos) admitindo assim que estas possam ser invocadas de formas alternativas, fornecendo, ou não, os valores para os parâmetros assim definidos. No exemplo acima, *employeeName* é o único parâmetro obrigatório.
- *self* corresponde ao *this* do Java, mas no Python é obrigatória a sua inclusão como primeiro parâmetro na definição dos métodos. (Na realidade a designação *self* é arbitrária, podendo ser usadas outras designações, mas, por convenção e clareza, deverá ser sempre usado *self*)
- Os métodos cujo nome começa e acaba com dois *underscore* (__), como o __init__() referido acima, são interpretados de forma especial. Por exemplo, o correspondente ao *toString()* do Java, é o método __str__(). Poderíamos definir este método na classe acima e alterar um pouco o exemplo anterior:

```
class Employee():
    def __init__(self, employeeName, taxDeductions=1, maritalStatus="single"):
         self.employeeName    = employeeName
         self.taxDeductions   = taxDeductions
         self.maritalStatus   = maritalStatus


    def __str__(self):
        return "{0},{2},{1}".format(self.employeeName,self.taxDeductions,self.maritalStatus)
```

```
>>> x.taxDeductions = 3
>>> for p in [x,y,z] :
        print(p)


João,single,3
Maria,single,2
Manuel,married,1
```

Uma boa introdução ao modo de concretizar mecanismos de herança em Python pode ser consultada em https://www.python-course.eu/python3_inheritance.php

Exercícios

1. Analise e compare as definições da classe *Point* em http://python4java.necaiseweb.org/OOP/OOP . Com base na definição anterior, implemente a classe `Ponto2D` que representa pontos 2D do tipo `(x,y)` utilizando os métodos seguintes:
   - `def __init__ (self, x = 0, y = 0)`
     Construtor.

- `def __str__(self)`
  Retorna uma `string` com a representação do objeto (equivalente ao `toString` em Java).

- `def distanciaOutroPonto (self, outroPonto)`
  Calcula a distância do objeto a outro ponto.

- `def distanciaOrigem (self)`
  Calcula a distância do objeto ao ponto (0,0) (deve reutilizar o método `distanciaOutroPonto`).

2. Adapte a resolução anterior de modo a criar uma classe *Ponto3D* para representar pontos num espaço tridimensional.

3. Implemente a classe *Circulo* utilizando os métodos seguintes:
   - `def __init__ (self, centro, raio)`
     Construtor
   - `def diametro (self)`
     Calcula o diâmetro.
   - `def area (self)`
     Calcula a área.