LINGUAGENS DE PROGRAMAÇÃO

Engenharia Informática

Programação Funcional

- Paradigma de programação
  - *Próximo das fundações matemáticas das ciências da computação*
  - *Composição de funções*
  - *Árvores de expressões – em vez de instruções sequênciais.*
    - Cada expressão devolve um valor
  - *As funções são tratadas como variávies (têm nomes atribuídos e podem ser passadas como argumentos ou retornadas.*

- Linguagem declarativa

Instruímos o computador de que resultado queremos

vs

Imperativa

Instruímos o computador passo a passo, como resolver o problema

exemplo

```
const numList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let result = 0;
for (let i = 0; i < numList.length; i++) {
    if (numList[i] % 2 === 0) {
        result += numList[i] * 10;
    }
}
```

```
const result = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
                .filter(n => n % 2 === 0)
                .map(a => a * 10)
                .reduce((a, b) => a + b);
```

em Python

- Python não é uma linguagem de programação funcional
- Incorpora alguns conceitos subjacentes.

- É normalmente codificado de forma imperativa, mas pode ser utilizado o estilo declarativo, quando apropriado.

Programação funcional - funcionalidades

- **Pure functions** – Não modificam o estado do programa. Dada uma entrada, produz sempre o mesmo resultado.
  - *Para atingir esta funcionalidade em Python: Não podemos modificar os argumentos, nem variáveis for a do scope da função (globais).*

```python
def multiply_2_pure(numbers):
    new_numbers = []
    for n in numbers:
        new_numbers.append(n * 2)
    return new_numbers

original_numbers = [1, 3, 5, 10]
changed_numbers = multiply_2_pure(original_numbers)
print(original_numbers) # [1, 3, 5, 10]
print(changed_numbers)  # [2, 6, 10, 20]
```

Programação funcional - funcionalidades

- **Imutabilidade** – Os dados não podem ser mudados depois de criados. P.ex: Se criarmos uma lista com 3 objetos e a considerarmos imutável, não devemos poder mudar/acrescentar/retirar items à lista. Mas podemos criar uma nova lista diferente.
  - *Para atingir esta funcionalidade em Python: Podemos utilizar Tuples em vez de Listas. Se tentarmos modificar o tuple, uma exceção é levantada.*

Programação funcional - funcionalidades

- **Funções de ordem superior** – Funções podem aceitar outras funções como parâmetros; e funções podem retornar outras funções como resultado.
  - *Permite criar uma abstração de ações: P.ex: se pretendemos passar uma função como evento de uma ação (carregar num botão) podemos ”entregar” uma função declarada num nível de hierarquia superior.*

```python
def write_repeat(message, n):
    for i in range(n):
        print(message)

write_repeat('Hello', 5)
```

Podemos passar a função como um parâmetro

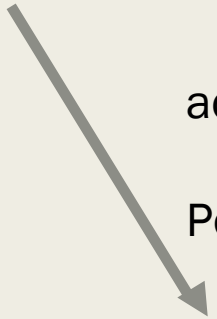
```python
def hof_write_repeat(message, n, action):
    for i in range(n):
        action(message)

hof_write_repeat('Hello', 5, print)

# Import the logging library
import logging
# Log the output as an error instead
hof_write_repeat('Hello', 5, logging.error)
```

```python
def add2(numbers):
    new_numbers = []
    for n in numbers:
        new_numbers.append(n + 2)
    return new_numbers

print(add2([23, 88])) # [25, 90]
```



add5 add10?

Podemos recriar funções

```python
def hof_add(increment):
    # Create a function that loops and adds the increment
    def add_increment(numbers):
        new_numbers = []
        for n in numbers:
            new_numbers.append(n + increment)
        return new_numbers
    # We return the function as we do any other value
    return add_increment

add5 = hof_add(5)
print(add5([23, 88]))   # [28, 93]
add10 = hof_add(10)
print(add10([23, 88]))  # [33, 98]
```

Expressões lambda

- São funções anónimas (sem nome ou identificador atribuído)
  - *Normalmente quando se declaram as funções com o **def**, existe um identificador associado à função.*
  - *Funcionalidade muito utilizada para criar funções de ordem superior.*

```
def hof_product(multiplier):
    return lambda x: x * multiplier

mult6 = hof_product(6)
print(mult6(6)) # 36
```

- Mais exemplos

```
lambda argument(s): expression
```

```
def remainder(num):
    return num % 2
```

```
remainder = lambda num: num % 2

print(remainder(5))
```

```
product = lambda x, y : x * y

print(product(2, 3))
```

Funções de ordem superior – built-in

- Algumas funções comummente utilizadas para operar sobre listas (iteráveis) já são suportadas em python.
  - *São funções que trabalham sobre listas e devolvem… novas listas (mais corretamente **iteradores** – para eficiência de memória)*
  - *map*
  - *filter*

- map
  - *Mapeia cada item da iteração num (novo) item*

```
map(object, iterable_1, iterable_2, ...)
```

```
names = ['Shivani', 'Jason', 'Yusef', 'Sakura']
greeted_names = map(lambda x: 'Hi ' + x, names)

# This prints something similar to: <map object at 0x10ed93cc0>
print(greeted_names)
# Recall, that map returns an iterator

# We can print all names in a for loop
for name in greeted_names:
    print(name)
```

- filter
  - *Testa cada item da iteração com uma função que devolve **True** ou **False**. No fim só os items com resultado **True** passam para o resultado.*

```
numbers = [13, 4, 18, 35]
div_by_5 = filter(lambda num: num % 5 == 0, numbers)

# We can convert the iterator into a list
print(list(div_by_5)) # [35]
```

- Juntar map e filter
  - *Cada função devolve um iterador e cada uma aceita iteráveis!*

```python
# Let's arbitrarily get the all numbers divisible by 3 between 1 and 20 and cube them
arbitrary_numbers = map(lambda num: num ** 3, filter(lambda num: num % 3 == 0, range(1, 21)))

print(list(arbitrary_numbers)) # [27, 216, 729, 1728, 3375, 5832]
```

E ainda...

- reduce
  - *Não está disponível buitl-in*
  - *import functools*
  - *Aplica uma operação matemática sobre todos os elementos e resulta **um valor** após iterar a lista toda.*

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num

# product = 24
```

1. Começa por aplicar a função matemática com os dois primeiros elementos.
2. De seguida, pega no resultado e aplica ao terceiro
3. Sucessivamente até ao fim
4. Devolve o valor final

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])

# Output: 24
```

Functional Programming vs Expression generators (list comprehension)

- O python não está pensado com a programação funcional em foco.
- Possui alguma abstração tal como se encontra noutras linguagens.
- A comunidade de desenvolvimento em Python tende a preferir a utilização de list comprehension, sempre que aplicável.

Fim

- Questões?