Aula 5
17/11/2025

UNIVERSIDADE LUSÓFONA

Fundamentos de Programação (T)

Licenciatura em Videojogos

Ano 1 / Semestre 1

Filipe Pinto
filipe.pinto@ulusofona.pt

UNIVERSIDADE LUSÓFONA

Sumário

- Funções como dados
- Lambdas
- Stacks
- Queues

UNIVERSIDADE LUSÓFONA

Funções como dados

UNIVERSIDADE LUSÓFONA

Funções como dados

- Em Python, funções são também dados

```
def say_hello():
    print("Hello World!")

print(say_hello)
```

Quando fazemos isto, estamos a definir uma variável chamada `say_hello` que contém um valor que é a função em si.

Reparem que não estamos a chamar a função `say_hello` – isso seria `say_hello()` – estamos só a pedir ao Python para nos escrever que valor tem a variável `say_hello`.

```
PS C:\Users\filip\Desktop\Videojogos\2526\FP\Aulas> & C:/Users/fil
n.exe c:/Users/filip/Desktop/Videojogos/2526/FP/Aulas/aula5_fp.py
<function say_hello at 0x000002C87ADC8A40>
```

UNIVERSIDADE LUSÓFONA

Funções como dados

- Como podemos usar as funções como dados?

```
def say_hello():
    print("Hello World!")

result = say_hello()
print(result)
```

O Python imprime o "Hello World" porque a função é chamada. Mas a variável `result` é nula.

```
PS C:\Users\filip\Desktop\Videojogos\2526\FP\Aulas>
ulas/aula5_fp.py
Hello World!
None
```

UNIVERSIDADE LUSÓFONA

Funções como dados

- Como é que resolvemos isto?

```
def say_hello():
    print("Hello World!")

result = say_hello()
print(result)
```

```
PS C:\Users\filip\Desktop\Vid
ulas/aula5_fp.py
Hello World!
None
```

```
def say_hello():
    return "Hello World!"

result = say_hello()
print(result)
```

```
PS C:\Users\filip\Desktop\V
ulas/aula5_fp.py
Hello World!
```

UNIVERSIDADE LUSÓFONA

Funções como dados

- O facto de funções serem dados dá imenso jeito...
- Exemplo:

```
def square(array_of_values):
    ret = []
    for value in array_of_values:
        ret.append(value ** 2)
    return ret

def double(array_of_values):
    ret = []
    for value in array_of_values:
        ret.append(value * 2)
    return ret

values = [1,2,3]

print(square(values))
print(double(values))
```

Apenas estas duas linhas são diferentes

```
PS C:\Users\filip\Desktop\Videojogos\2526
ulas/aula5_fp.py
[1, 4, 9]
[2, 4, 6]
```

UNIVERSIDADE LUSÓFONA

Funções como dados

- Neste exemplo temos duas funções praticamente idênticas
- O que significa que podemos otimizar ainda mais o código
- Se funções são dados, podemos escrever uma função que recebe uma outra função como parâmetro
- Assim podemos preencher as listas sem replicar código
- Para isso vamos definir primeiro as funções utilitárias:

```
def square(value):
    return value ** 2

def double(value):
    return value * 2
```

UNIVERSIDADE LUSÓFONA

Funções como dados

- E agora vamos definir a função que processa os dados:
- Tinhamos este código inicial:
- E agora passamos a ter isto:

```python
def square(array_of_values):
    ret = []
    for value in array_of_values:
        ret.append(value ** 2)
    return ret

def double(array_of_values):
    ret = []
    for value in array_of_values:
        ret.append(value * 2)
    return ret
```

```python
def process_values(array_of_values, process_function):
    ret = []
    for value in array_of_values:
        ret.append(process_function(value))
    return ret
```

Recebemos a função como uma variável normal

E usamos essa variável como se fosse uma função

Funções como dados

- E agora basta chamar a função:

```
values = [1,2,3]

print(process_values(values, square))
print(process_values(values, double))
```

UNIVERSIDADE LUSÓFONA

Funções como dados

- O exemplo completo

```python
def square(value):
    return value ** 2

def double(value):
    return value * 2

def process_values(array_of_values, process_function):
    ret = []
    for value in array_of_values:
        ret.append(process_function(value))
    return ret

values = [1,2,3]

print(process_values(values, square))
print(process_values(values, double))
```

```
PS C:\Users\filip\Desktop\Videojogos\2526\FP\Aulas>
ulas/aula5_fp.py
[1, 4, 9]
[2, 4, 6]
```

UNIVERSIDADE LUSÓFONA

Funções como dados

- Num dos exercícios da ultima aula tinhamos um sobre criar uma calculadora em que poderiamos ter usado esta lógica

```python
# Operações
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        print("Error: division by zero!")
        return None
    return a / b
```

Definiamos as operações

E usavamos essas funções como variáveis

```python
# Selecção
def calculate(a, b, operation):
    if operation == "add":
        return add(a, b)
    elif operation == "subtract":
        return subtract(a, b)
    elif operation == "multiply":
        return multiply(a, b)
    elif operation == "divide":
        return divide(a, b)
    else:
        print("Unknown operation.")
        return None # equivalente a null em outras linguagens
```

UNIVERSIDADE LUSÓFONA

Exercícios

UNIVERSIDADE LUSÓFONA

Exercícios

```
execute_action(attack)
execute_action(defend)
execute_action(cure)
execute_action(inventory)
execute_action(flee)
```

```
You attacked the enemy!
You raised your shield!
You gained 10 HP!
You opened your inventory.
You fleed combat!
```

1. Cria um sistema que executa as ações: ataque, defesa, cura, abrir inventário e fugir.
    - Usa uma função para gerir as ações.
2. Usa uma lista `actions = []` para guardar as funções executá-las como no código do exercício anterior

```
You attacked the enemy!
You attacked the enemy!
You gained 10 HP!
You fleed combat!
You fleed combat!
```

3. Define uma função que executa a mesma acção duas vezes usando o código do exercício anterior
4. Cria uma função que, conforme parametro recebido retorna uma função diferente. Depois, chama a função de return
5. Usando um dicionário. Cria um menu para executar o código todo do exercício usando input do jogador.
    - Verifica se a opção existe
    - Usa funções de string para limpar o input

```
menu = {
    "a": attack,
    "d": defend,
    "c": cure,
    "i": inventory,
    "f": flee
}
```

```
Select command a/d/c/i/f: a
You attacked the enemy!

Select command a/d/c/i/f: b

invalid option


Select command a/d/c/i/f: c
You gained 10 HP!

Select command a/d/c/i/f: d
You raised your shield!

Select command a/d/c/i/f: f
You fled combat!
PS C:\Users\filip\Desktop\Videojogos\2526\FP\Aulas>
```

Lambdas

UNIVERSIDADE LUSÓFONA

Lambdas

- São também chamados de "funções anónimas" ou "*closures*"
- Isto é uma função com um nome:

```
def square(value):
    return value ** 2
```

- Isto é um lambda:

```
lambda value : value ** 2
```

- Podemos também dar um nome a um lambda

```
square = lambda value : value ** 2
```

Lambdas

- Diferenças em Python:
- Uma lambda não precisa de return, está implícito

```
square = lambda value : value ** 2
print("lambda result:", square(2))
```

```
PS C:\Users\filip\Desktop\Vide
ython312/python.exe c:/Users/f
lambda result: 4
```

- Uma lambda só pode ter uma instrução
- Pode ser usado como uma expressão:

```
print(process([1,2,3], lambda value : -value
```

- Isto é uma expressão, vamos ver como a podemos utilizar.

UNIVERSIDADE LUSÓFONA

Lambdas

- Temos uma função base `process()`:

```python
def process(lista, func):
    resultado = []
    for value in lista:
        resultado.append(func(value))
    return resultado
```

Esta função que recebe tem dois parâmetros. Recebe uma lista e outra função que processa a lista

- Queremos processar uma lista para devolver números negativos
- Podemos escrever uma nova função:

```python
def turn_negative(value):
    return -value

print(process([1, 2, 3], turn_negative))
```

- Ou usar um lambda:

```python
print(process([1, 2, 3], lambda value: -value))
```

```
PS C:\Users\filip\Desktop\Videojogos\2
ython312/python.exe c:/Users/filip/Des
[-1, -2, -3]
```

Lambdas

- Podemos também usar lambas dentro de ciclos:

```
square = lambda value : value ** 2

list = [1,2,3,4,5]


for i in list:

    print("Square of", i, ":", square(i))
```

```
PS C:\Users\filip\Desktop\Videojogos\25
ython312/python.exe c:/Users/filip/Deskt
Square of 1 : 1
Square of 2 : 4
Square of 3 : 9
Square of 4 : 16
Square of 5 : 25
```

Lambdas

- Podemos também usar expressões condicionais:
- Sintaxe → `lambda value: return_if_true <condição> else return_if_false.`

```
evaluation = lambda grade: "passed" if grade >= 9.5 else "failed"

grades = [5, 10, 18, 9, 14, 8]

for g in grades:

    print("Grade:", g, "->", evaluation(g))
```

```
Grade: 5 -> failed
Grade: 10 -> passed
Grade: 18 -> passed
Grade: 9 -> failed
Grade: 14 -> passed
Grade: 8 -> failed
```

UNIVERSIDADE LUSÓFONA

Lambdas

- Podemos também usar expressões que usem multiplos dados
- Neste caso vamos usar uma lista de tuplos:

```
clientes = [("none", 100), ("normal", 100), ("premium", 100), ("vip", 100)]

def desconto(tipo):
    if tipo == "normal":
        return lambda valor: valor * 0.95
    elif tipo == "premium":
        return lambda valor: valor * 0.90
    elif tipo == "vip":
        return lambda valor: valor * 0.80
    else:
        return lambda valor: valor

for tipo, preco in clientes:
    print("Cliente:", tipo, "| Valor final:", desconto(tipo)(preco))
```

```
Cliente: none | Valor final: 100
Cliente: normal | Valor final: 95.0
Cliente: premium | Valor final: 90.0
Cliente: vip | Valor final: 80.0
```

É o equivalente a isto:

```
funcao_desconto = desconto(tipo)  # devolve a lambda
valor_final = funcao_desconto(preco)  # lambda com o preço
print("Cliente:", tipo, "| Valor final:", valor_final)
```

UNIVERSIDADE LUSÓFONA

Lambdas

- São úteis para tornar código mais legível, ou menos propenso a erros
- Também Podemos reduzir a quantidade copy & paste que se faz
- Podemos utilizar lambdas na nossa aventura de texto

Lambdas

- Este é o código actual

```
elif move_player("north", NORTH, -1, 0):
    pass
elif move_player("south", SOUTH, 1, 0):
    pass
elif move_player("east", EAST, 0, 1):
    pass
elif move_player("west", WEST, 0, -1):
    pass
```

- O código como está é dificil de ler a não ser que se veja a função move_player

- Como é que poderiamos aplicar lambdas aqui?

Lambdas

- Criamos 4 lambdas e damos-lhe um nome

```
move_north = lambda : move_player("north", NORTH, -1, 0)
move_south = lambda : move_player("south", SOUTH, 1, 0)
move_east = lambda : move_player("east", EAST, 0, 1)
move_west = lambda : move_player("west", WEST, 0, -1)
```

Lambdas

- Passamos a ter o código assim, o que é mais simples e fácil de ler:

```
elif (move_north()):
    pass
elif (move_south()):
    pass
elif (move_east()):
    pass
elif (move_west()):
```

Lambdas

- Assim o código fica mais simples

  e mantém a funcionalidade

```
(2, 1) : H
What now?
Choose the direction you want to go north, south, east, west) or exit
North

You move north

Room description: C

(2, 0) : C
What now?
Choose the direction you want to go north, south, east, west) or exit
east

You move east

Room description: D

(3, 0) : D
What now?
Choose the direction you want to go north, south, east, west) or exit
west

You move west

Room description: C

(2, 0) : C
What now?
Choose the direction you want to go north, south, east, west) or exit
south

You move south
```

Lambdas

- E temos assim o código:

```
position = (2, 2)

command = ""

def move_player(direction_text, direction_index, y_inc, x_inc):
    global position

    x, y = position
    if command == direction_text:
        if room_exits[y][x][direction_index]:
            print("\nYou move " + direction_text)
            y += y_inc
            x += x_inc
            position = (x, y)
            print("\nRoom description: "  + room_descriptions[y][x] + "\n")
        else:
            print("You can't move " + direction_text + "!")
        return True
    return False

move_north = lambda : move_player("north", NORTH, -1, 0)
move_south = lambda : move_player("south", SOUTH, 1, 0)
move_east = lambda : move_player("east", EAST, 0, 1)
move_west = lambda : move_player("west", WEST, 0, -1)

while command != "exit":
    x, y = position
    print(position, ":", room_descriptions[y][x])
    print("What now?")

    command = input("Choose the direction you want to go north, south, east, west) or exit\n").lower()
    command = command.strip()

    if command == "":
        continue
    elif command == "exit":
        print("Exiting game...")
        continue
    elif (move_north()):
        pass
    elif (move_south()):
        pass
    elif (move_east()):
        pass
    elif (move_west()):
        pass
    else:
        print("I don't understand " + command + "!")
```

UNIVERSIDADE LUSÓFONA

Mas podemos melhorar ainda mais....

- Podemos combinar várias técnicas baseando-nos nas últimas aulas
- Primeiro vamos criar um dicionário para processar todos os comandos.

```
command_processor = {
    "north" : move_north,
    "south" : move_south,
    "east"  : move_east,
    "west"  : move_west
}
```

UNIVERSIDADE LUSÓFONA

Mas podemos melhorar ainda mais....

- Podemos combinar várias técnicas baseando-nos nas últimas aulas
- Primeiro vamos criar um dicionário:
- Agora podemos substituir o código antigo:

```
command_processor = {
    "north" : move_north,
    "south" : move_south,
    "east"  : move_east,
    "west"  : move_west
}
```

```
elif (move_north()):
    pass
elif (move_south()):
    pass
elif (move_east()):
    pass
elif (move_west()):
```

UNIVERSIDADE LUSÓFONA

Mas podemos melhorar ainda mais....

- Podemos combinar várias técnicas baseando-nos nas últimas aulas
- Primeiro vamos criar um dicionário:

```
command_processor = {
    "north" : move_north,
    "south" : move_south,
    "east"  : move_east,
    "west"  : move_west
}
```

- Agora podemos substituir o código antigo:

```
elif (move_north()):
    pass
elif (move_south()):
    pass
elif (move_east()):
    pass
elif (move_west()):
```

Por isto

```
if (command in command_processor):
    command_processor[command]()
elif command == "exit":
    print("Exiting game...")
    continue
else:
    print("I don't understand " + command + "!")
```

Mas podemos melhorar ainda mais....

- Agora que usamos dicionários e lambdas podemos facilmente introduzir comandos simplificados como abreviaturas:

```
command_processor = {
    "north" : move_north,
    "n" : move_north,
    "south" : move_south,
    "s" : move_south,
    "east"  : move_east,
    "e"  : move_east,
    "west"  : move_west,
    "w"  : move_west
}
```

No entanto...

Se corrermos o código...

- As abreviaturas não estão a executar os comandos
- Mas também não devolvem "`invalid command`"

Porquê?

```
(2, 2) : M
What now?
Choose the direction you want to go north, south, east, west) or exit
n
(2, 2) : M
What now?
Choose the direction you want to go north, south, east, west) or exit
north

You move north

Room description: H

(2, 1) : H
What now?
Choose the direction you want to go north, south, east, west) or exit
s
(2, 1) : H
What now?
Choose the direction you want to go north, south, east, west) or exit
south

You move south

Room description: M

(2, 2) : M
What now?
Choose the direction you want to go north, south, east, west) or exit
```

UNIVERSIDADE LUSÓFONA

Mas podemos melhorar ainda mais....

- Isto acontece porque temos código redundante que se sobrepõe ao novo código que introduzimos

O nosso código já faz a verificação do `direction_text`; em: `if(command in command_processor)`

No entanto a função `move_player()` só dá `return True` se o `command == direction_text`, caso contrário dá `return False`

Por isso as nossas abreviaturas dão `return False` embora sejam reconhecidas no while como um comando válido

```
def move_player(direction_text, direction_index, y_inc, x_inc):
    global position

    x, y = position
    if command == direction_text:
        if room_exits[y][x][direction_index]:
            print("\nYou move " + direction_text)
            y += y_inc
            x += x_inc
            position = (x, y)
            print("\nRoom description: "  + room_descriptions[y][x] + "\n")
        else:
            print("You can't move " + direction_text + "!")
        return True
    return False
```

UNIVERSIDADE LUSÓFONA

Mas podemos melhorar ainda mais....

- A remoção essas 3 linhas da função irá

  dar funcionalidade às abreviaturas:

```
def move_player(direction_text, direction_index, y_inc, x_inc):
    global position

    x, y = position
    if room_exits[y][x][direction_index]:
        print("\nYou move " + direction_text)
        y += y_inc
        x += x_inc
        position = (x, y)
        print("\nRoom description: "  + room_descriptions[y][x] + "\n")
    else:
        print("You can't move " + direction_text + "!")
```

```
(2, 2) : M
What now?
Choose the direction you want to go north, south, east, west) or exit
n

You move north

Room description: H

(2, 1) : H
What now?
Choose the direction you want to go north, south, east, west) or exit
s

You move south

Room description: M

(2, 2) : M
What now?
Choose the direction you want to go north, south, east, west) or exit
e
You can't move east!
(2, 2) : M
What now?
Choose the direction you want to go north, south, east, west) or exit
w
You can't move west!
(2, 2) : M
```

UNIVERSIDADE LUSÓFONA

Exercícios

UNIVERSIDADE LUSÓFONA

Exercícios

1. Cria um dicionário de lambdas que calcula o dano final de acordo com o tipo de ataque.
   - Variável → `dano_base = 40`
   - Todas as armas têm um multiplier. Espada (1.2), Arco (1.1), Magia (1.5)

```
Arma: espada | Dano final: 48.0
Arma: arco | Dano final: 44.0
Arma: magia | Dano final: 60.0
```

2. Cria um lambda que receba os pontos de vida e devolva "vivo" se forem maiores que 0, ou "morto" caso contrário.
   - Lista: `vida = [50, 10, 0, -5]`

```
HP: 50 -> vivo
HP: 10 -> vivo
HP: 0 -> morto
HP: -5 -> morto
```

3. Cria uma função `pontuacao(tipo)` que devolve uma lambda diferente dependendo do tipo de evento.
   - Usa essa função para calcular pontuação
   - Usa uma lista de tuplos para as acções e valores: `acoes = [("inimigo", 50), ("chefe", 100), ("missao", 20)]`
   - Cada lambda tem também pontuação extra -> Inimigo *2; Chefe *5, Missão +100; Outro + 0.
   - Imprime a pontução final

```
Ação: inimigo | Valor base: 50 | Pontuação final: 100
Ação: chefe | Valor base: 100 | Pontuação final: 500
Ação: missao | Valor base: 20 | Pontuação final: 120
Ação: objetivo | Valor base: 10 | Pontuação final: 10
```