# Programação I
# Aula 4 — Definição de procedimentos e funções

Pedro Vasconcelos
DCC/FCUP



## Nesta aula...

# Definição de procedimentos e funções

- Na aula passada vimos algumas funções pré-definidas na módulo `math`
- Nesta aula: vamos ver como definir novos procededimentos e funções

**Programação estruturada**

Decompor um problema em procedimentos mais simples até chegar às operações elementares.



# Definições

```
def nome(lista de parâmetros):
    primeira instrução
    segunda instrução
    ⋮
    instrução final
```

- A *lista de parâmetros* pode ser vazia
- O *corpo* consiste de uma ou mais instruções
  - delimitadas pela *indentação*
  - alinhadas numa mesma coluna

# Exemplo

```
def refrão():
    print("Se um elefante incomoda muita gente,")
    print("Dois elefantes incomodam muito mais.")
```

Experimentando no interpretador:

```
>>> refrão()
Se um elefante incomoda muita gente,
Dois elefantes incomodam muito mais.
```



# Observações

- Um procedimento é uma receita para uma computação
- A *definição* dum procedimento não executa as instruções (apenas regista o procedimento)

```
def refrão():    # definição do procedimento
    ⋮
```

- As instruções são executadas quando *invocamos* o procedimento

```
>>> refrão()     # invocar o procedimento
```

- Podemos invocar um procedimento dentro de outro; exemplo:

```
def repetir():
    refrão()    # primeira invocação
    refrão()    # segunda invocação
```

# Parâmetros e valores

Vamos generalizar o procedimento anterior.

```
def refrão(n):
    print("Se", n, "elefantes incomodam muita gente,")
    print(n+1, "elefantes incomodam muito mais.")
```

A variável *n* é um parâmetro do procedimento.

Para invocar o procedimento temos de especificar o valor de *n*:

```
>>> refrão(3)
Se 3 elefantes incomodam muita gente,
4 elefantes incomodam muito mais.
```



# Observações

Ao invocar o procedimento temos de usar o número correto de argumentos.

```
>>> refrão()
TypeError: refrão() missing 1 required positional
argument: 'n'
>>> refrão(2,3)
TypeError: refrão() takes 1 positional argument
but 2 were given
```

# Observações (cont.)

Considere esta variação do procedimento anterior:

```
def refrão(n):
    print("Se n elefantes incomodam muita gente,")
    print("n+1 elefantes incomodam muito mais.")
```

(Notar a colocação das aspas.)

O que acontece ao executar?

```
>>> refrão(2)
```

(Experimente no interpretador!)



# Definir funções

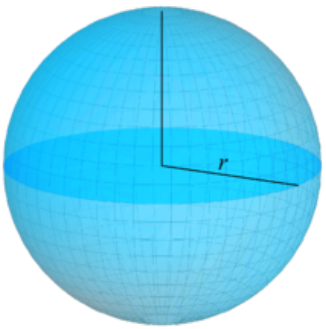
```
def função(arg1, arg2, ...):
    ⋮
    return resultado
```

- Uma função é um procedimento que retorna um resultado
- A última instrução de uma função deve ser `return`
- A instrução `return` pode também ser usada no meio do corpo (para terminar a função)

# Exemplo: calcular o volume de uma esfera

Volume $V$ de uma esfera de raio $r$.

$$V = \frac{4}{3}\pi r^3$$




```
import math   # para usar aproximação a π
def volume(r):
    V = 4/3 * math.pi * r**3
    return V
```



# Exemplo: calcular o volume de uma esfera (cont.)

Exemplos de uso:

```
>>> volume(1.0)
4.1887902047863905
>>> volume(1.5)
14.137166941154067
>>> volume(2.0)
33.510321638291124
```

# Definição alternativa

```
import math
def volume(r):
    return 4/3 * math.pi * r**3
```

- Podemos retornar o valor de uma *expressão* diretamente
- Não é necessário usar a variável auxiliar *V*
- Contudo: usar variáveis auxiliares pode ajudar a documentar o programa



# Return ou print?

```
def volume(r):
    V= 4/3*math.pi*r**3
    return V

>>> volume(1)+volume(2)
37.69911184307752
```

```
def volume(r):
    V= 4/3*math.pi*r**3
    print(V)

>>> volume(1)+volume(2)
4.188790204786390 5
33.510321638291124
TypeError:...
```

- A definição usando `print` não retorna o resultado
- É preferível definir procedimentos que retornam resultados
- Permite usar o resultado para outros fins (além de imprimir)

# Âmbito das variáveis

- Os parâmetros de um procedimento são *variáveis locais*
  - são associadas a valores apenas dentro da definição
  - não temos de nos preocupar se esse nome é usado noutro contexto
- As variáveis auxiliares definidas dentro de procedimentos são também locais

```
>>> r = 42
>>> volume(1)
4.1887902047863905
>>> r
42
>>> V
NameError: name 'V' is not defined
```



# Documentação

Podemos documentar procedimentos e funções usando

1. comentários
2. *docstrings*

```
import math   # usar definições matemáticas
def volume(r):
    "Calcula o volume de uma esfera dado o raio r."
    V= 4/3*math.pi*r**3
    return V
```

# Documentação (cont.)

- As *docstrings* são usadas pelo sistema de documentação de Python
- Usamos `help` para pedir ajuda sobre um módulo ou procedimento:

  `>>> help(`*nome*`)`
- Colocar comentários e *docstrings* ajuda o programador a compreender um programa ou biblioteca

```
>>> help(volume)
Help on function volume in module __main__:

volume(r)
    Calcula o volume de uma esfera de raio r.
```



# Anotações de tipos

- Os argumentos de procedimentos e funções devem respeitar tipos certos
- Exemplo: o argumento do procedimento `refrão` deve ser inteiro (o número *n* de elefantes)
- Podemos documentar essa informação usando uma anotação de tipo

```
def refrão(n: int):
   print("Se", n, "elefantes incomodam muita gente,")
   print(n+1, "elefantes incomodam muito mais.")
```

# Anotações de tipos (cont.)

Para funções, anotamos os tipos dos argumentos e do resultado.

```
def volume(r: float) -> float:
    V= 4/3*math.pi*r**3
    return V
```

Mais geralmente:

```
def função(x: tipo1, y: tipo2, ...) -> tipoR:
    ⋮
    return resultado
```



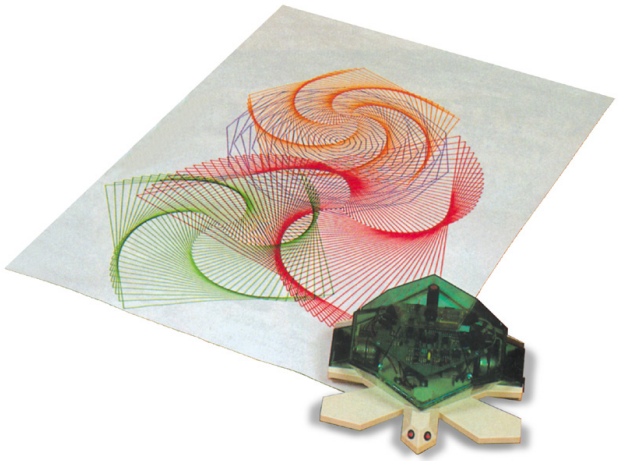
# Anotações de tipos (cont.)

- As anotações de tipos são opcionais em Python
- Os programas executam sem quaisquer anotações
- Contudo: as anotações permitem mais facilmente detetar erros lógicos nos programas
- O sistema de testes automáticos usa-as para dar mensagens de erros mais informativas

# Módulo turtle

Vamos fazer programas que desenham usando o módulo *turtle*:

- o programa controla um *robot* virtual (tartaruga)
- desloca-se para frente, para trás e roda sobre si próprio
- usa uma caneta para pintar um rasto
- muito simples, mas permite fazer desenhos sofisticados





# Primeiros passos

Devemos começar por importar o módulo:

```
>>> import turtle
```

Os comandos têm a forma `turtle.`*`comando(...)`*:

```
>>> turtle.clear()         # limpar a janela
>>> turtle.forward(100)    # avançar 100 pixels
>>> turtle.left(90)        # rodar 90 graus à esquerda
>>> turtle.forward(200)    # avançar 200 pixels
```

# Primeiros passos (cont.)

Em alternativa, podemos usar

```
>>> from turtle import *
```

e omitir o nome do módulo:

```
>>> clear()
>>> forward(100)
>>> left(90)
>>> forward(200)
```



# Comando principais

| | |
|---|---|
| forward(*n*) | avançar *n pixels* |
| backward(*n*) | retroceder *n pixels* |
| left(*a*) | rodar *a* graus à esquerda |
| right(*a*) | rodar *a* graus à direita |
| color(*c*) | mudar a cor do traço |
| pensize(*n*) | mudar a largura do traço |
| penup() | levantar a caneta |
| pendown() | baixar a caneta |
| speed(*n*) | mudar a velocidade da tartaruga |
| clear() | limpar a janela |
| reset() | limpar a janela e re-inicializar a tartaruga |

# Desenhar um quadrado

Vamos definir um procedimento para desenhar um quadrado com 100 *pixels* de lado.

- Desenhar quatro lados, rodando 90° para a esquerda após cada lado
- Alternativa: poderiamos rodar para a direita



# Desenhar um quadrado (cont.)

```
def quadrado():
    forward(100)  # primeiro lado
    left(90)
    forward(100)  # segundo lado
    left(90)
    forward(100)  # terceiro lado
    left(90)
    forward(100)  # quarto lado
    left(90)      # termina na orientação original
```

# Evitando repetições

- Repetimos quatro vezes duas instruções:

```
forward(100)
left(90)
```

```
forward(100)
left(90)
```

```
forward(100)
left(90)
```

```
forward(100)
left(90)
```

- Podemos evitar a repetição de código usando um ciclo



# Evitando repetições (cont.)

```
def quadrado():
    for lado in range(4): # repetir 4 vezes
        forward(100)
        left(90)
```

- Efetuamos um *ciclo* sobre uma lista de valores
- O *corpo do ciclo* executa com a variavel `lado` tomando os valores 0, 1, 2, 3
- O objetivo é apenas repetir 4 vezes; poderiamos ter usado outros valores

# Variantes

```
def quadrado():
    for i in range(4): # 0, 1, 2, 3
        forward(100)
        left(90)

def quadrado():
    for i in [1,2,3,4]:
        forward(100)
        left(90)

def quadrado():
    for c in ['red','green','blue','black']:
        color(c)
        forward(100)
        left(90)
```



# Generalizando

- Vamos generalizar o procedimento para desenhar quadrado com lado qualquer
- Basta tomar como parâmetro a medida do lado (inteiro)

```
def quadrado(lado: int):
    "Desenhar um quadrado com lado dado."
    for i in range(4):
        forward(lado)
        left(90)
```

# Exemplo maior: uma espiral

```
from turtle import *

def quadrado(lado: int):
    for c in ['red','blue','green','black']:
        color(c)
        forward(lado)
        left(90)

reset()
speed(10)
for i in range(36):   # desenhar 36 quadrados
    quadrado(50+i*5)  # com lados crescentes
    left(10)          # ângulo entre os quadrados
```