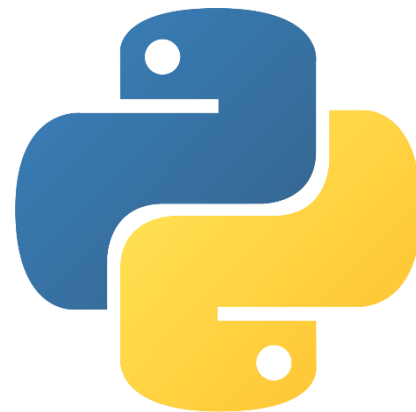


Curso de Iniciação à Programação em Python para Profissionais de Saúde



Luís Vieira e José Ferrão

Instituto Nacional de Saúde Doutor Ricardo Jorge

17-26 Março 2025

Sessão 6: Números aleatórios

- **Conteúdos:**
 - Programas determinísticos vs. aleatórios
 - Números aleatórios
 - Ciclo *while*
 - Boas práticas de construção de programas
 - Exercícios
 - Jogo da vida
 - Documentação do *Python*

Programas determinísticos

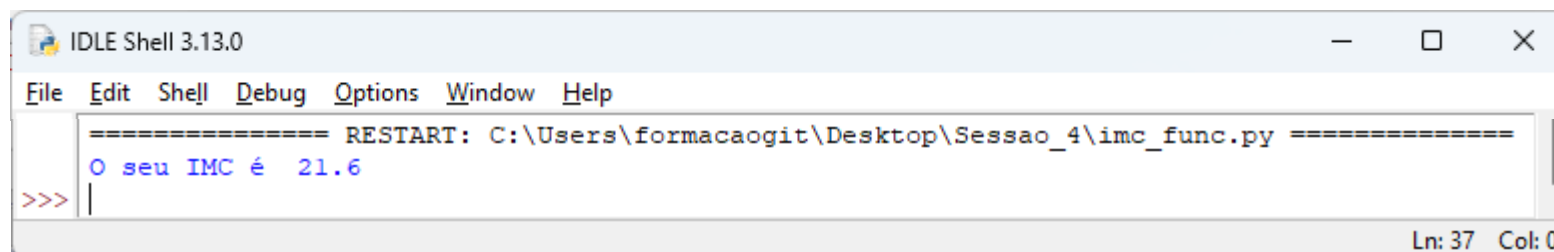
- Um programa diz-se **determinístico** se produz o mesmo resultado cada vez que é corrido com os mesmos dados de partida. Por exemplo, se a função *massa_corp* for invocada como `massa_corp(70, 1.80)`, produzirá sempre o mesmo resultado:

```
def massa_corp (p, a):
```

```
    """Esta função calcula o índice de massa corporal (IMC) usando os valores
    de peso (p) e altura (a). O IMC é calculado usando a fórmula  $IMC = p/a^{**2}$ ."""
```

```
    IMC = p / a**2
```

```
    print ("O seu IMC é", round(IMC, 2))
```



```
IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
===== RESTART: C:\Users\formacaogit\Desktop\Sessao_4\imc_func.py =====
O seu IMC é 21.6
>>>
Ln: 37 Col: 0
```

Programas estocásticos

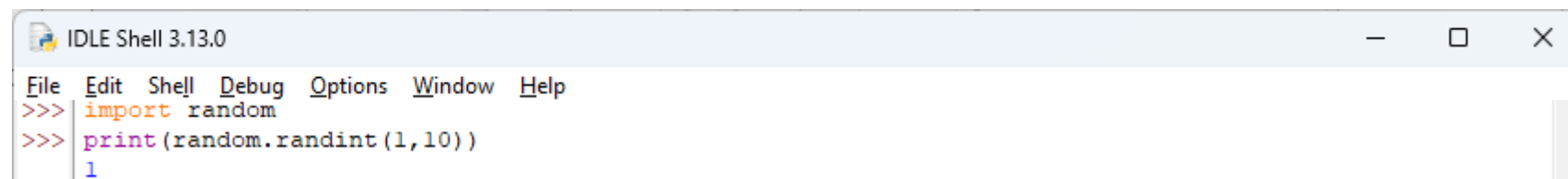
- Um processo estocástico é aquele em que o próximo estado depende dos estados anteriores e de algum elemento aleatório
- Os programas **estocásticos** são aqueles que incorporam a aleatoriedade para gerarem o resultado
- Um programa estocástico tem como objectivo produzir um bom resultado, mas não garante a obtenção do melhor resultado possível
- Em muitas linguagens de programação, incluindo o *Python*, existem funções que podem gerar valores “aleatórios”

Módulo *random*

- O *Python* possui um módulo designado **random** que, tal como o nome indica, foi desenhado para gerar **números ou letras aleatórios**
- O módulo **random** possui várias funções e é muito utilizado na programação em *Python*, particularmente em modelos de simulação
- Tal como os outros módulos do *Python*, o módulo **random** tem de ser importado antes de se poder usar qualquer das suas funções:

```
>>> import random
```

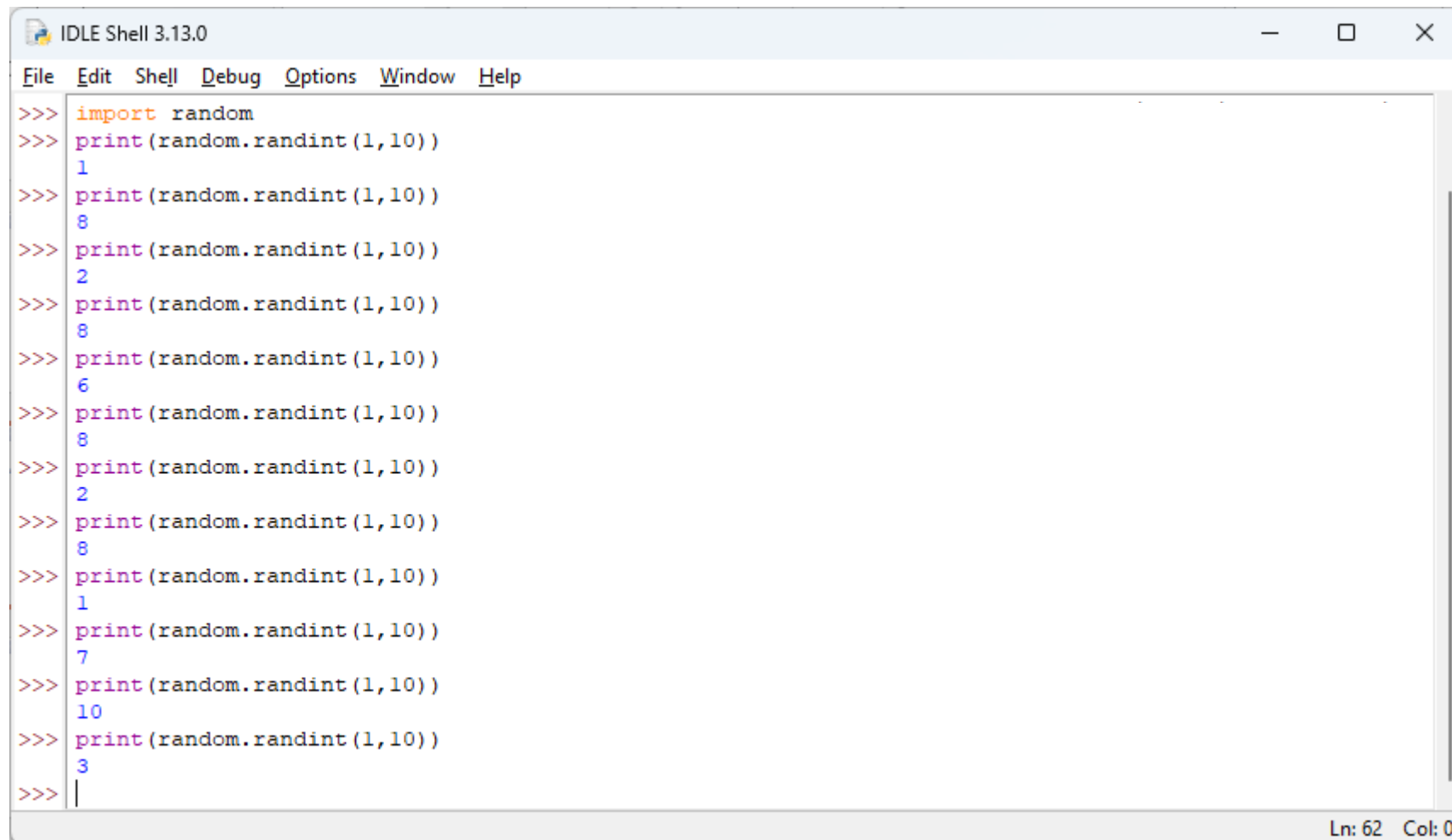
- Uma das funções (*randint()*) deste módulo permite gerar números inteiros aleatórios dentro de um determinado intervalo. Por exemplo, usando a função *randint()* com o intervalo [1;10], obtém-se o número 1:



```
IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
>>> import random
>>> print(random.randint(1,10))
1
```

Números aleatórios

- Se usarmos a mesma função várias vezes consecutivas, com o mesmo intervalo de valores, obteremos resultados diferentes



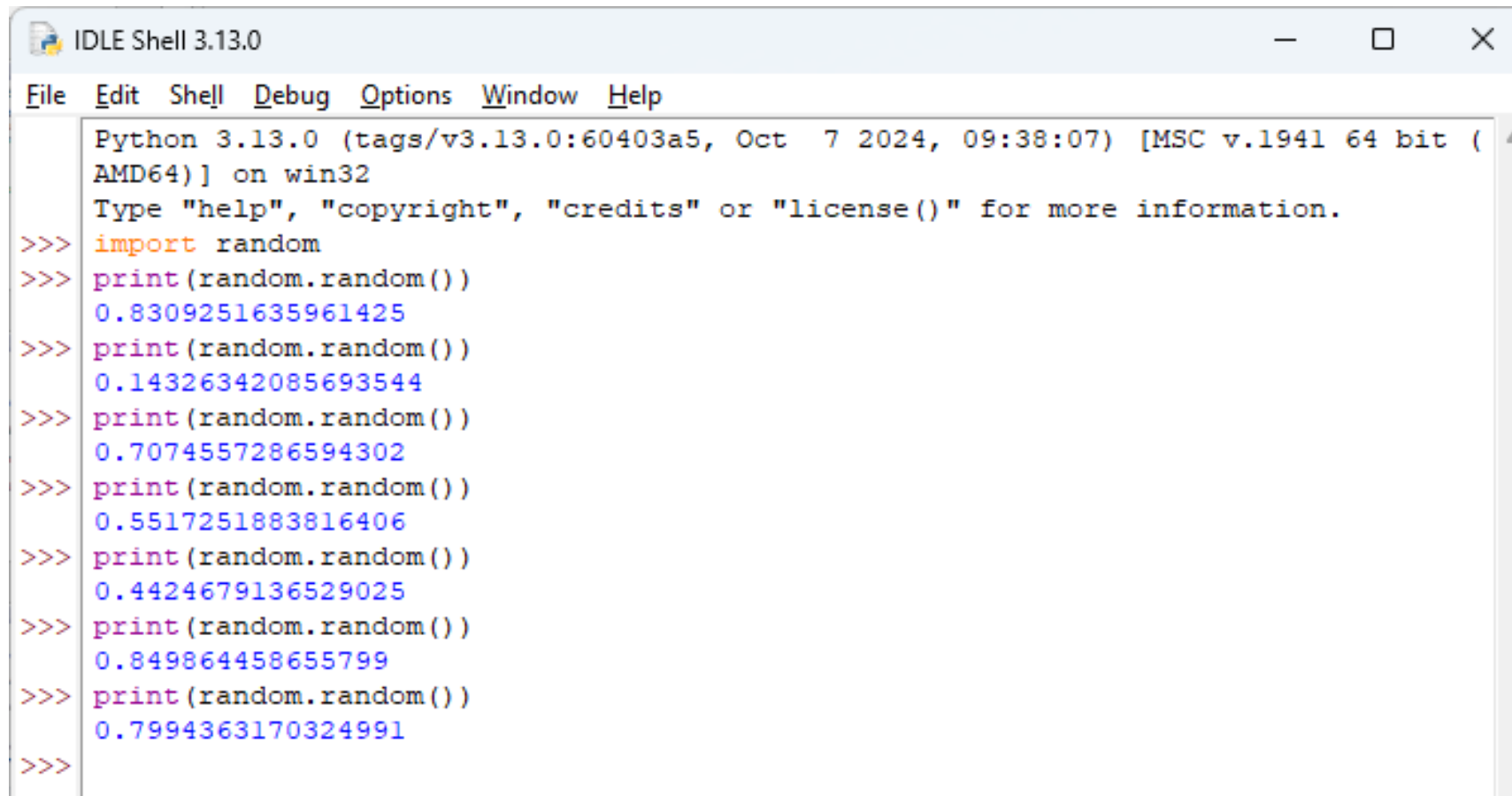
```

IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
>>> import random
>>> print(random.randint(1,10))
1
>>> print(random.randint(1,10))
8
>>> print(random.randint(1,10))
2
>>> print(random.randint(1,10))
8
>>> print(random.randint(1,10))
6
>>> print(random.randint(1,10))
8
>>> print(random.randint(1,10))
2
>>> print(random.randint(1,10))
8
>>> print(random.randint(1,10))
1
>>> print(random.randint(1,10))
7
>>> print(random.randint(1,10))
10
>>> print(random.randint(1,10))
3
>>>
  
```

Ln: 62 Col: 0

Números aleatórios

- O módulo **random** também contém a função *random()*, que permite gerar números aleatórios do tipo *float* no intervalo entre 0.0 e 1.0

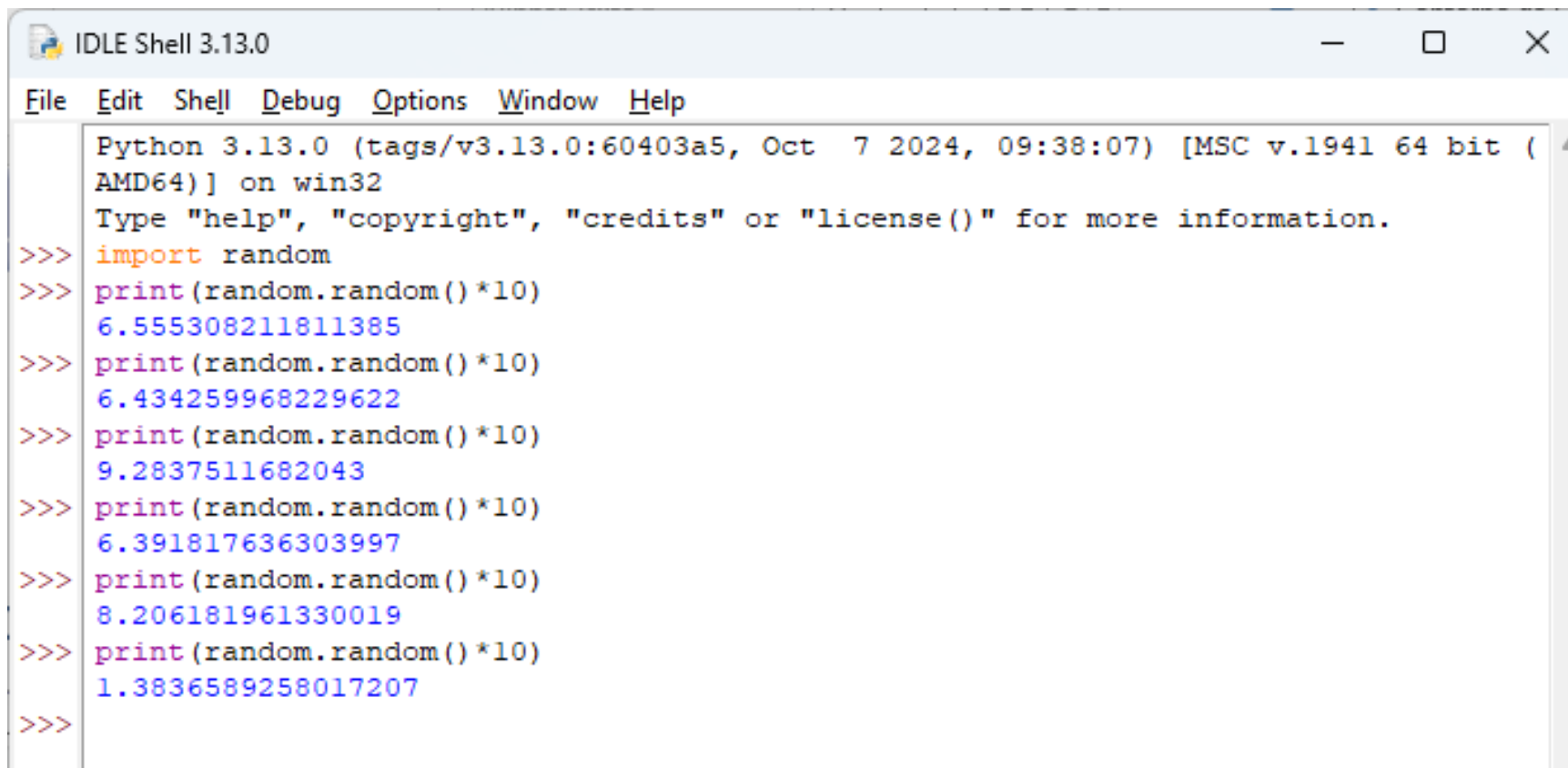


```

IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import random
>>> print(random.random())
0.8309251635961425
>>> print(random.random())
0.14326342085693544
>>> print(random.random())
0.7074557286594302
>>> print(random.random())
0.5517251883816406
>>> print(random.random())
0.4424679136529025
>>> print(random.random())
0.849864458655799
>>> print(random.random())
0.7994363170324991
>>>
  
```


Números aleatórios

- Usando a função *random()*, também é possível produzir *floats* aleatórios no intervalo [1;10] através da multiplicação do resultado da função por 10

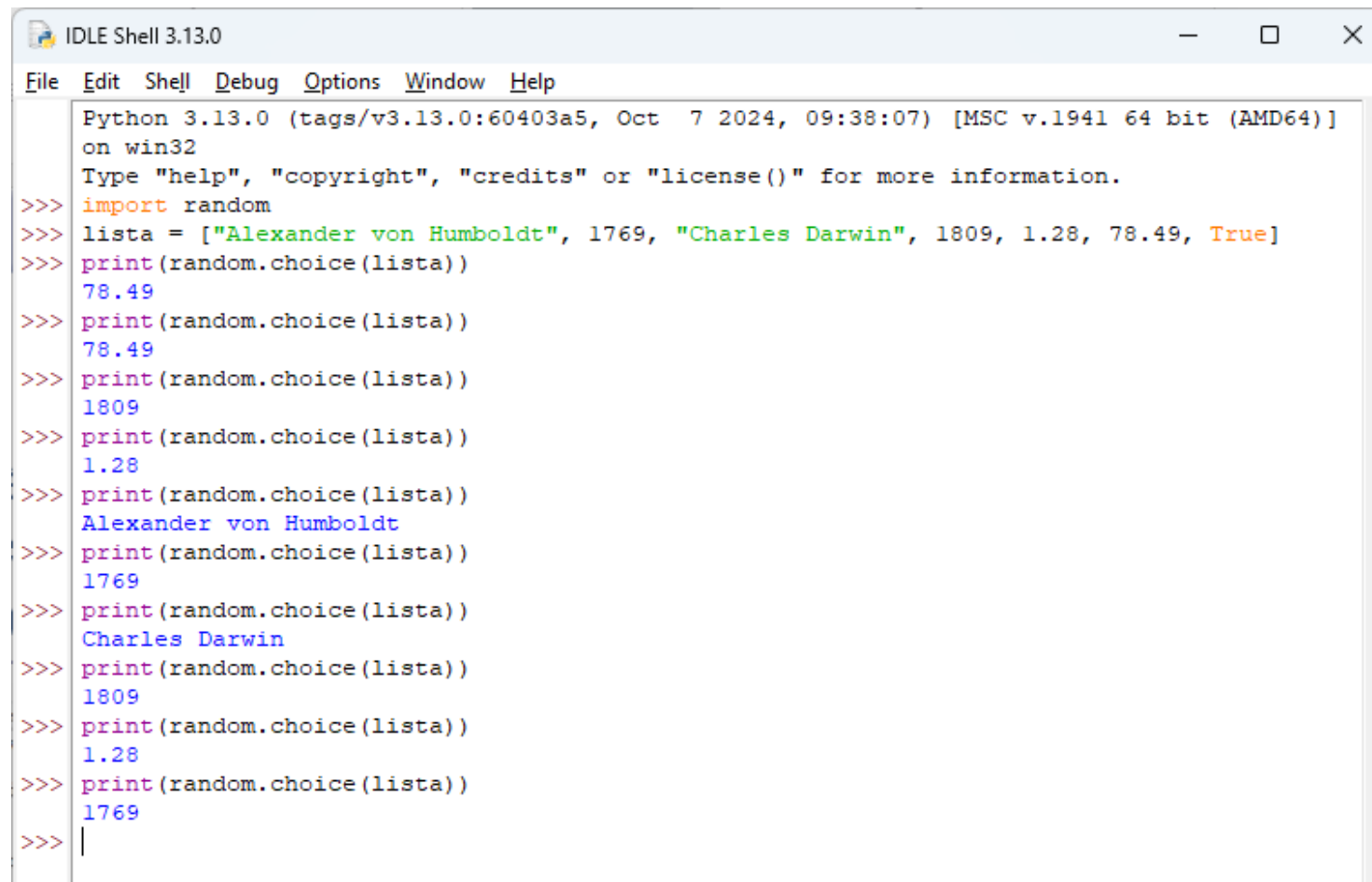


```

IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import random
>>> print(random.random()*10)
6.555308211811385
>>> print(random.random()*10)
6.434259968229622
>>> print(random.random()*10)
9.2837511682043
>>> print(random.random()*10)
6.391817636303997
>>> print(random.random()*10)
8.206181961330019
>>> print(random.random()*10)
1.3836589258017207
>>>
  
```


Números aleatórios

- O módulo **random** não é utilizado exclusivamente para números. Por exemplo, a função *choice()* permite escolher um item aleatoriamente de uma lista que contenha *strings*, valores booleanos, *floats* e/ou números inteiros



```

IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import random
>>> lista = ["Alexander von Humboldt", 1769, "Charles Darwin", 1809, 1.28, 78.49, True]
>>> print(random.choice(lista))
78.49
>>> print(random.choice(lista))
78.49
>>> print(random.choice(lista))
1809
>>> print(random.choice(lista))
1.28
>>> print(random.choice(lista))
Alexander von Humboldt
>>> print(random.choice(lista))
1769
>>> print(random.choice(lista))
Charles Darwin
>>> print(random.choice(lista))
1809
>>> print(random.choice(lista))
1.28
>>> print(random.choice(lista))
1769
>>>
  
```

Números aleatórios

- A posição dos itens de uma lista também pode ser alterada aleatoriamente usando a função *shuffle()* antes da função *choice()*, por forma a aumentar a aleatoriedade da escolha dos itens

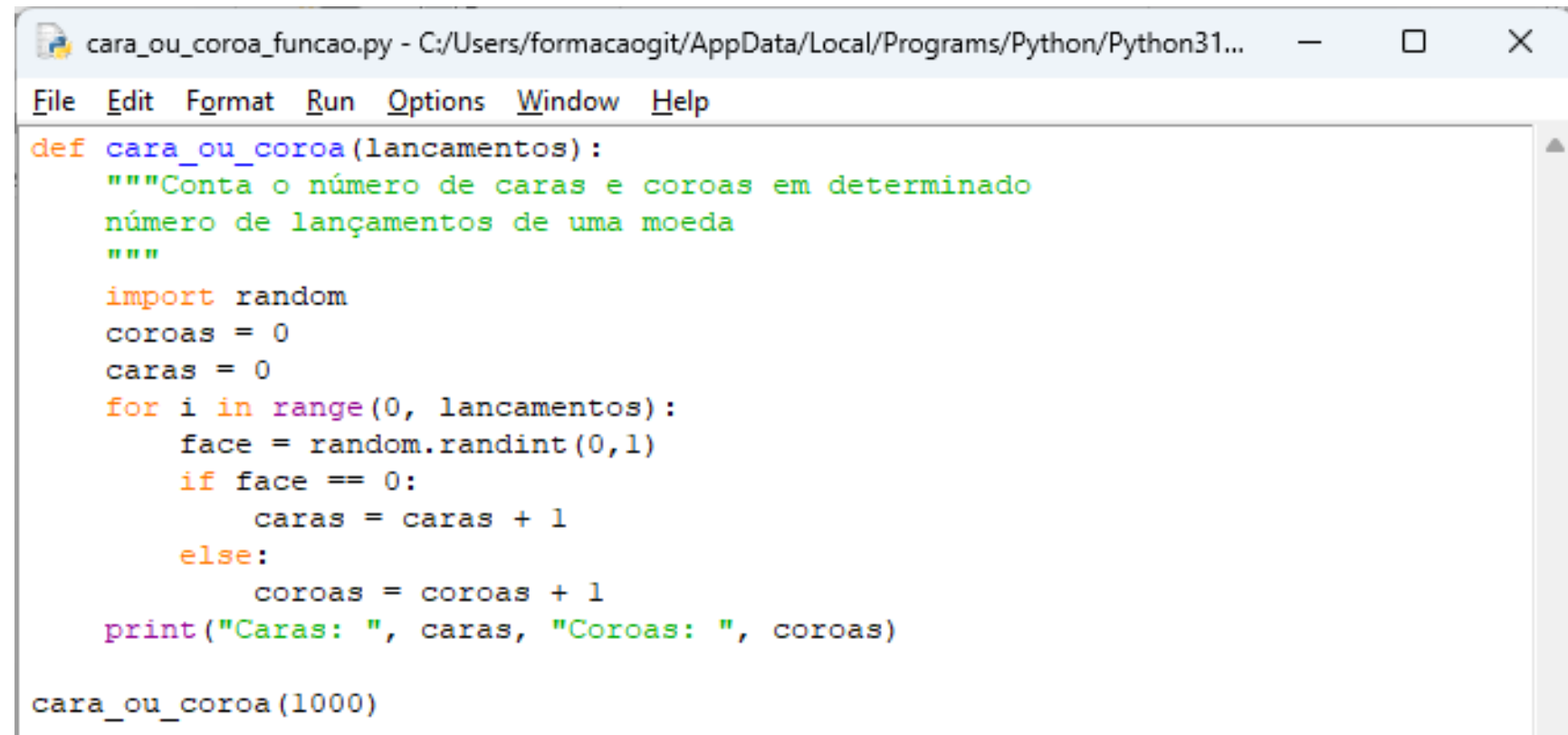
```

IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import random
>>> lista = ["Alexander von Humboldt", 1769, "Charles Darwin", 1809, 1.28, 78.49, True]
>>> random.shuffle(lista)
>>> print(lista)
['Charles Darwin', 'Alexander von Humboldt', 1.28, True, 1769, 1809, 78.49]
>>> print(random.choice(lista))
78.49
>>> random.shuffle(lista)
>>> print(lista)
[True, 78.49, 'Charles Darwin', 'Alexander von Humboldt', 1769, 1.28, 1809]
>>> print(random.choice(lista))
True
>>> random.shuffle(lista)
>>> print(lista)
['Charles Darwin', 1809, 1.28, 1769, 'Alexander von Humboldt', True, 78.49]
>>> print(random.choice(lista))
1809
>>>

```

Números aleatórios

- Um exemplo simples de um programa que gera números aleatórios é o que simula lançamentos de uma moeda ao ar e conta o número de caras e coroas. A função `cara_ou_coroa` é invocada usando como argumento o número de lançamentos (l)



```

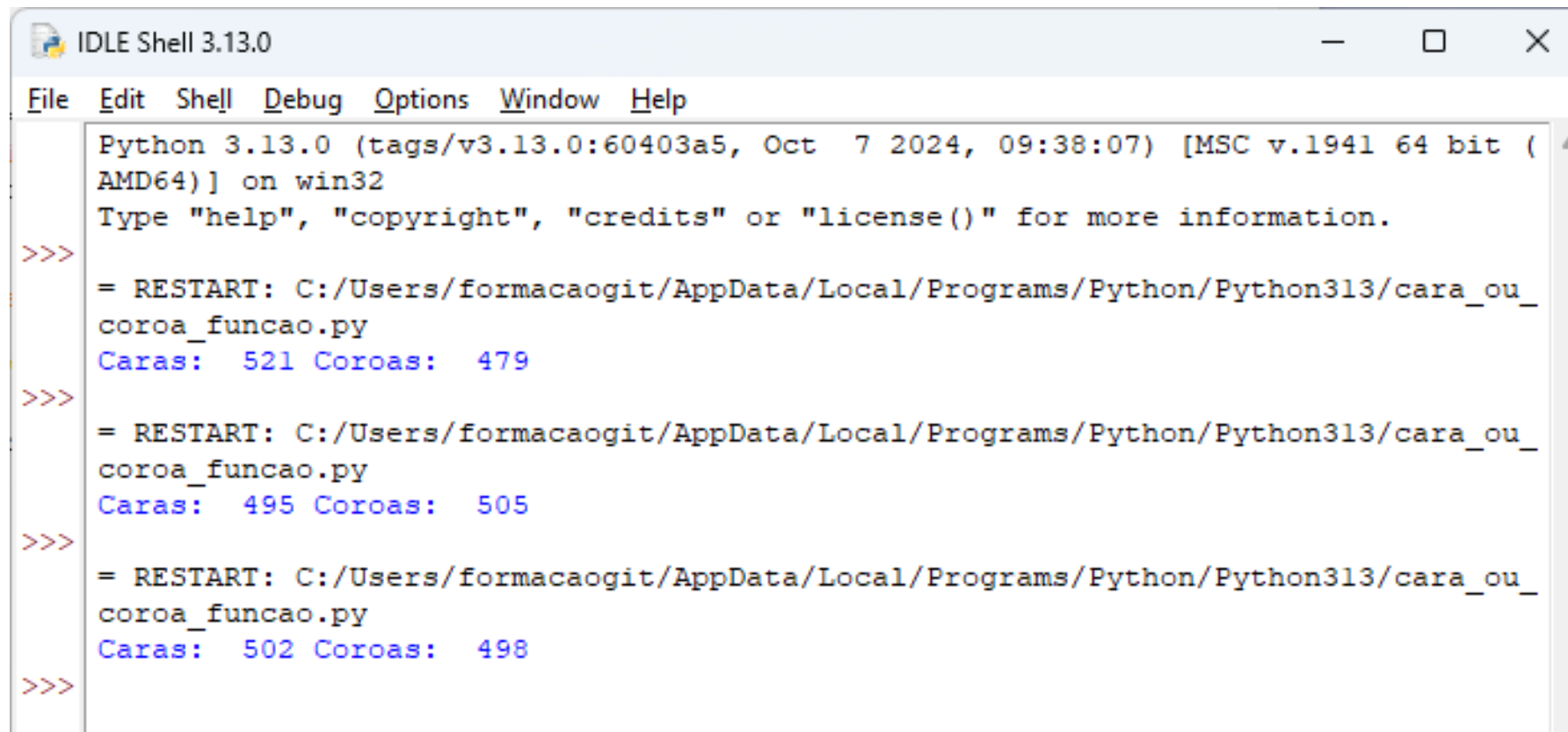
cara_ou_coroa_funcao.py - C:/Users/formacaogit/AppData/Local/Programs/Python/Python31...
File Edit Format Run Options Window Help
def cara_ou_coroa(lancamentos):
    """Conta o número de caras e coroas em determinado
    número de lançamentos de uma moeda
    """
    import random
    coroas = 0
    caras = 0
    for i in range(0, lancamentos):
        face = random.randint(0,1)
        if face == 0:
            caras = caras + 1
        else:
            coroas = coroas + 1
    print("Caras: ", caras, "Coroas: ", coroas)

cara_ou_coroa(1000)

```

Números aleatórios

- A invocação da função `cara_ou_coroa` com argumento `lancamentos = 1000`, produz resultados distintos de cada vez, mas que se aproximam do resultado esperado (500 caras e 500 coroas)

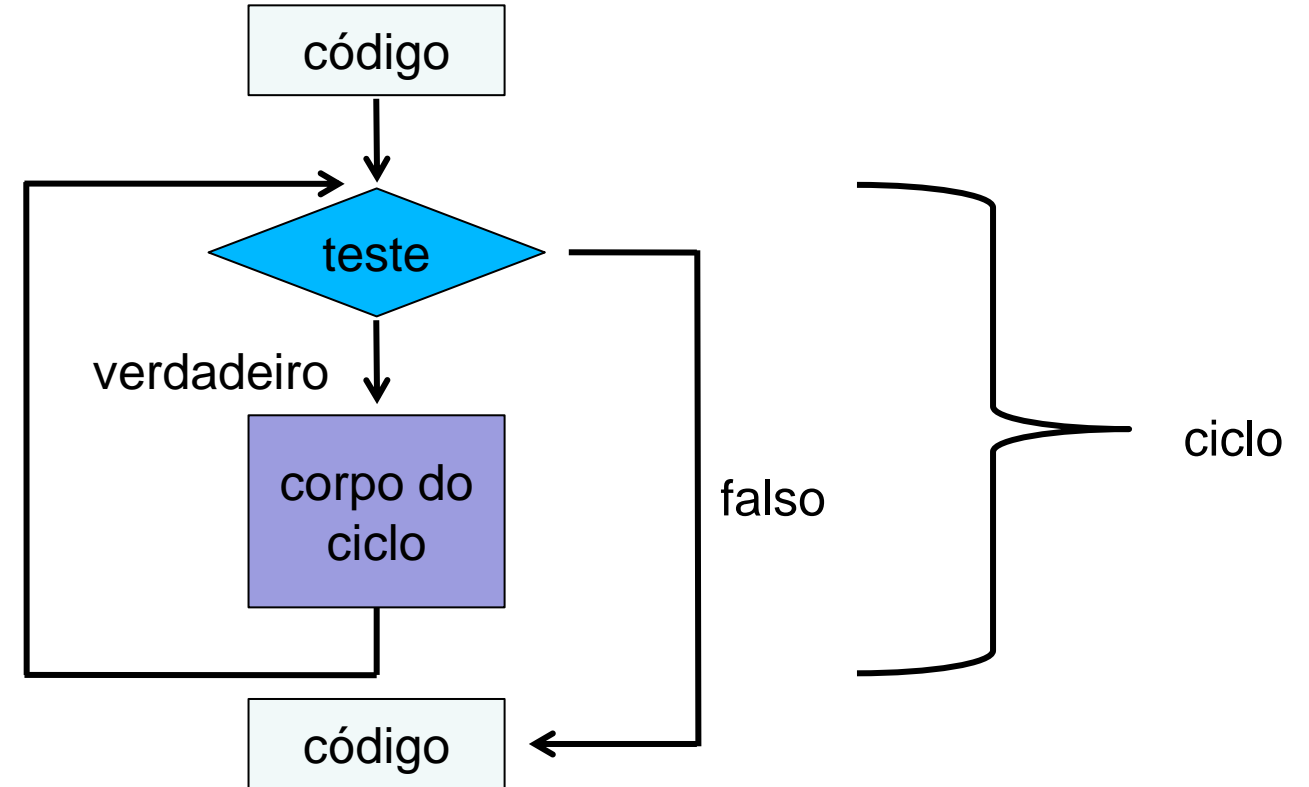


```

IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/formacaogit/AppData/Local/Programs/Python/Python313/cara_ou_coroa_funcao.py
Caras: 521 Coroas: 479
>>>
= RESTART: C:/Users/formacaogit/AppData/Local/Programs/Python/Python313/cara_ou_coroa_funcao.py
Caras: 495 Coroas: 505
>>>
= RESTART: C:/Users/formacaogit/AppData/Local/Programs/Python/Python313/cara_ou_coroa_funcao.py
Caras: 502 Coroas: 498
>>>
  
```

Ciclo *while*

- O ciclo *while* é outro processo de iteração do *Python* que começa com um teste condicional
- Se o teste é avaliado como verdadeiro (*True*), o programa executa o corpo do ciclo uma vez e depois repete o teste
- O corpo do ciclo é repetido até que o teste seja avaliado como falso (*False*)
- Nesta situação, a execução do código passa para a secção após o ciclo *while*



Ciclo *while*

- O ciclo **while** tem a seguinte estrutura:

while condição_de_teste:

código_do_corpo_do_ciclo

função_decrescente (opcional)

- Uma vez que o ciclo só termina quando a condição_de_teste é avaliada como *False*, a(s) variável(eis) do corpo do ciclo podem assumir múltiplos valores. Assim, é útil escrever numa folha de papel o valor de cada variável em cada passo do ciclo, para perceber o seu comportamento
- Em alguns ciclos **while**, é recomendado usar-se uma **função decrescente** que permita limitar o número de iterações do ciclo por forma a que este não corra “para sempre”

Ciclo *while*

- Usemos como exemplo o seguinte código, que gera múltiplos de um número inteiro positivo introduzido pelo utilizador. O ciclo **while** não termina se a função decrementadora não for incluída no corpo do ciclo

```

multiplos.py - C:/Users/formacaogit/AppData/Local/Programs/Python/Python313/multiplos....
File Edit Format Run Options Window Help
# Este script produz x-1 múltiplos de um valor inteiro de x

x = int(input("Introduza um valor inteiro > 0: "))

multiplo = 0
iteracoes = x

print("int:", "mul:", "iter:")

while iteracoes != 0:
    multiplo = multiplo + x
    iteracoes = iteracoes - 1 # função decrescente
    print("\n" + str(x) + "    " + str(multiplo) + "    " + str(iteracoes))
    
```

```

IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/formacaogit/AppData/Local/Programs/Python/Python313/multiplo
s.py
Introduza um valor inteiro > 0: 4
int: mul: iter:

4      4      3

4      8      2

4      12     1

4      16     0
>>>
    
```


Boas práticas de construção de programas

- Conhecer as declarações condicionais, ciclos, funções, etc. do *Python* é crucial para se aprender a programar, mas saber por onde começar a escrever um novo programa pode ser uma tarefa difícil
- A primeira coisa a fazer é ler e **interpretar cuidadosamente o problema**, identificando todos os tipos de dados que são descritos e, em simultâneo, separar os que são dados de entrada (*input*) e de saída (*output*)
- O reconhecimento dos tipos de dados envolvidos, e do seu processamento, pode permitir **identificar “sub-problemas”** dentro do problema maior a resolver (desenvolvimento *top-down*)
- Após terem sido identificados os vários “sub-problemas”, deve focar-se a atenção em cada um deles separadamente, nomeadamente através da criação de uma função que permita resolver esse “sub-problema”

Boas práticas de construção de programas

- Em cada função, devem identificar-se os argumentos necessários, as variáveis a usar no corpo da função e o “conhecimento específico” (por ex., uma expressão matemática específica), e combinar tudo para produzir o resultado desejado
- Nesta fase, deve **testar-se várias vezes a função**, e comparar com o resultado obtido por um processo independente, a fim de detectar eventuais erros antes de o programa estar completo
- Se há um problema com a função, uma boa forma de perceber onde está o problema consiste em usar a **função *print*** nas variáveis em vários passos da função
- Repetir o processo para cada “sub-problema” e, no final, combinar as várias funções para responder ao problema inicial

Exercícios

- Gregor Mendel (1822-1884) foi um monge que se dedicou ao estudo da variação em plantas, tendo ficado conhecido como o 'pai' da genética. Numa das suas experiências, Mendel cruzou plantas que produziam unicamente flores de cor branca com plantas que produziam unicamente flores de cor roxa. As plantas resultantes deste cruzamento (1ª geração ou F1) apresentaram apenas flores de cor roxa. De seguida, Mendel efectuou a auto-fertilização destas plantas e verificou que na geração seguinte (2ª geração ou F2), 705 plantas tinham flores roxas e 224 plantas apresentavam flores brancas (proporção aproximada de 3:1). Se Mendel tivesse analisado apenas 50 plantas resultantes da 2ª geração teria chegado a esta mesma conclusão?

Construa uma função que utilize números aleatórios para simular os resultados da 2ª geração de plantas e calcule, em função do número total de plantas estudadas (por ex., 50, 1000 ou 10000), a proporção das mesmas que apresenta flores brancas. Corra o programa várias vezes para observar a variação nos resultados obtidos.

Exercícios

- Suponhamos que Mendel antecipava que cerca de $\frac{1}{4}$ das plantas resultantes da 2ª geração seriam brancas, mesmo antes de produzirem flores. Crie uma função que use um ciclo *while* para calcular o número mínimo de plantas que Mendel teria de seleccionar aleatoriamente, de entre as plantas da geração F2, para confirmar a hipótese de que a proporção de plantas de cor branca era de 0.25.

Corra o programa várias vezes para observar a variação nos resultados obtidos. Qual é o número mais elevado de plantas que obteve na simulação e que confirma a hipótese de Mendel?

Nota: O interessante desta simulação é que, em teoria, poderíamos nunca atingir o número de plantas necessário para obter a proporção de 0.25!

Jogo da vida

- O **jogo da vida** (*Game of Life*) foi criado pelo matemático britânico John Conway em 1970, com o objectivo de simular a evolução de um grupo de células ou seres vivos ao longo do tempo (autómato celular)
- Um **autómato celular** é um modelo computacional de evolução temporal que permite exibir comportamentos complexos
- No jogo da vida não existem jogadores, ou seja, a evolução é apenas determinada pelo estado inicial, não necessitando de mais nenhuma intervenção
- O utilizador interage com o jogo da vida criando uma configuração inicial e observando como esta evolui ao longo do tempo
- O jogo da vida desenvolve-se num tabuleiro bidimensional em que a “unidade” de jogo é uma grelha quadrada de 9 células, isto é, cada célula tem 8 células vizinhas
- Cada célula pode assumir apenas 2 estados distintos: viva (“On”) ou morta (“Off”)

Jogo da vida

- O jogo da vida tem as seguintes **4 regras**:
 - Qualquer célula viva com menos de 2 vizinhos vivos morre de solidão
 - Qualquer célula viva com mais de 3 vizinhos vivos morre de superpopulação
 - Qualquer célula morta com exactamente 3 vizinhos vivos torna-se uma célula viva
 - Qualquer célula viva com 2 ou 3 vizinhos vivos continua no mesmo estado para a geração seguinte
- Todos os nascimentos e mortes ocorrem simultaneamente, ou seja, constituem uma geração ou um “instante” na história evolutiva das células/seres vivos
- O estado de cada geração serve de base para determinar o estado de todas as células/seres vivos na geração seguinte

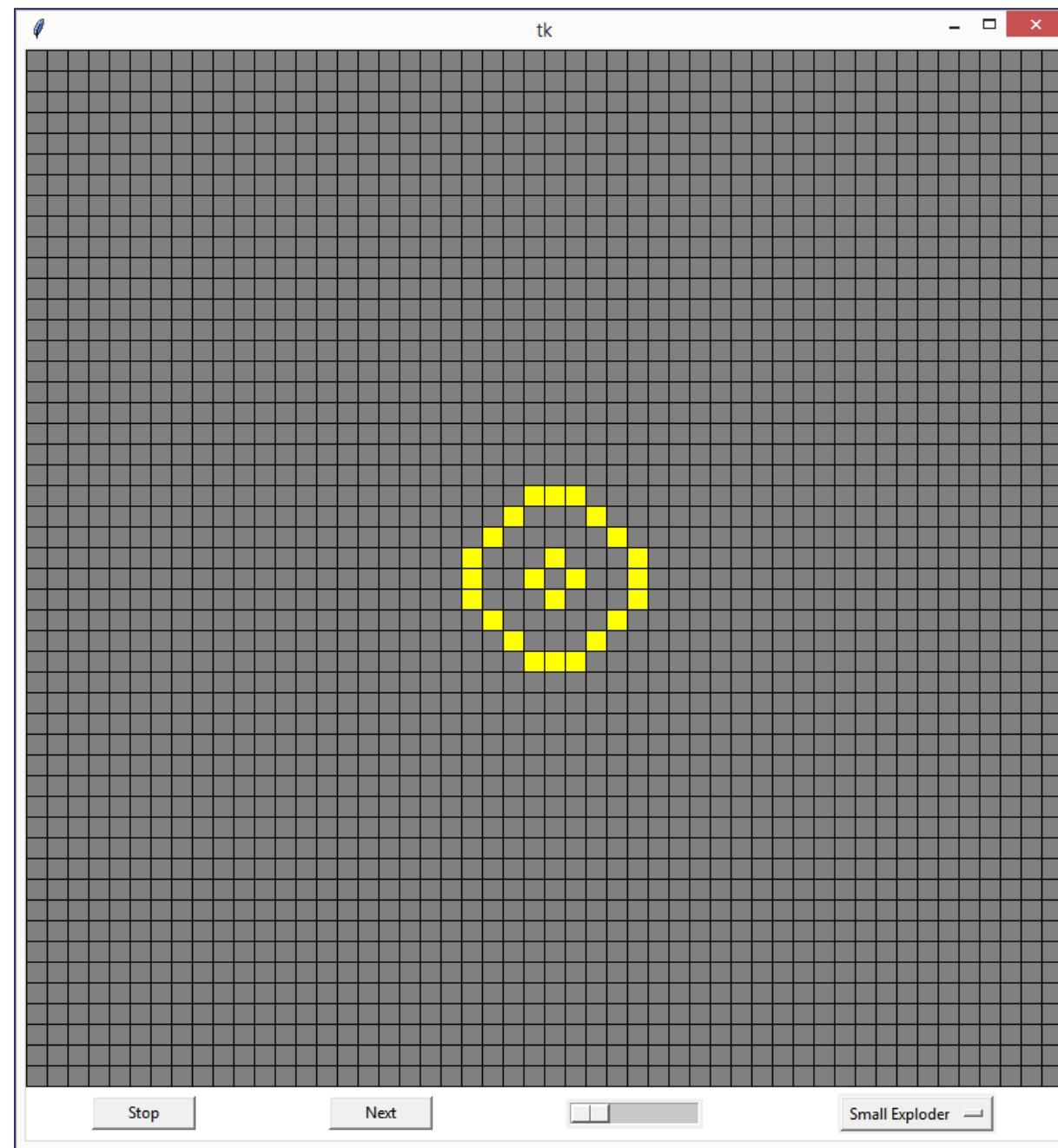
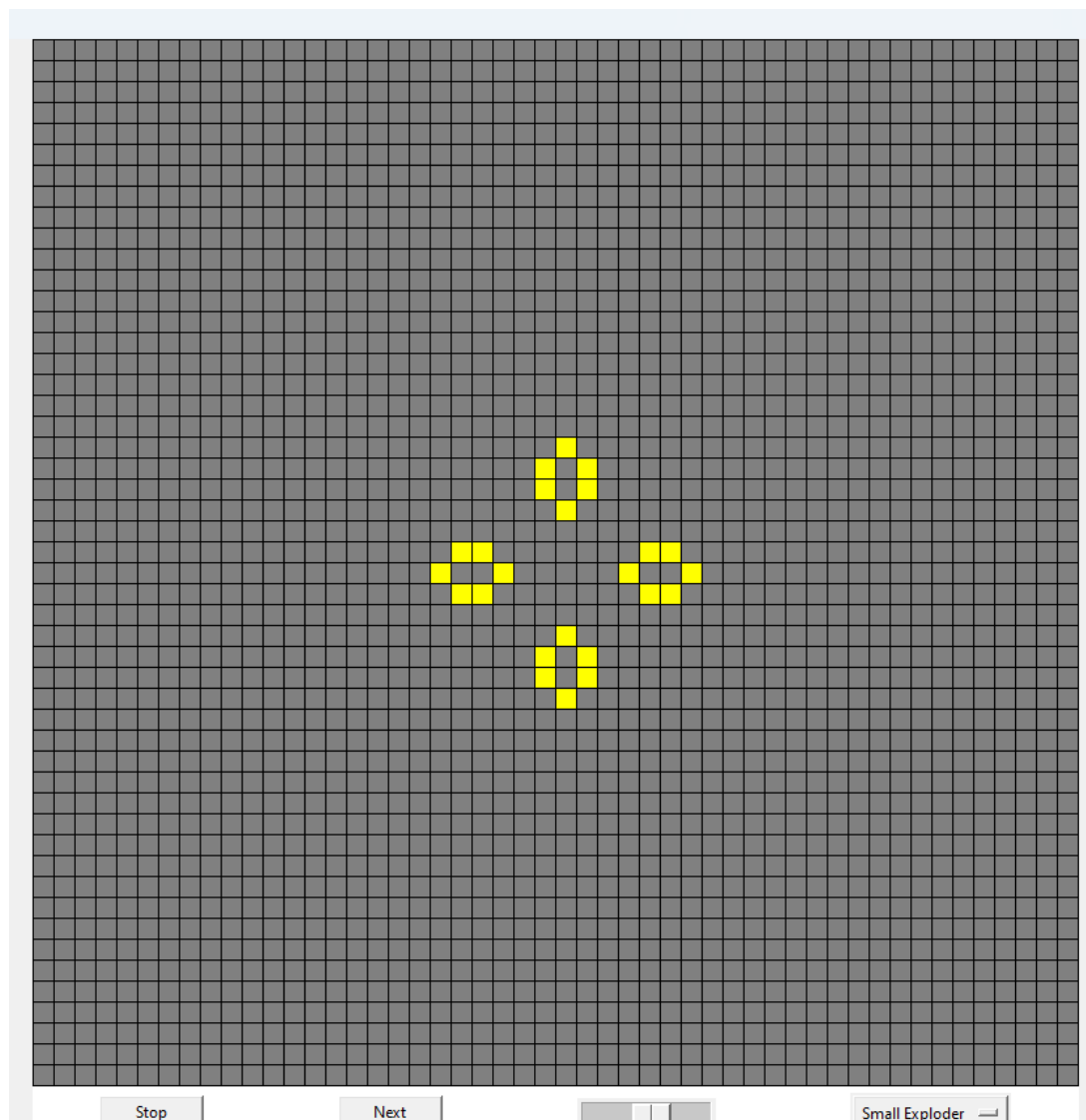
Jogo da vida

- Para visualizar o jogo da vida, comecemos por instalar uma implementação simples do jogo, existente no *Python Package Index* e que só utiliza as bibliotecas *built-in* do *Python*. Usando a linha de comandos do Windows, escrever:

> *pip install game-of-life-nodeps*
- Para correr o programa, aceder ao directório onde se encontram os *scripts* do *Python* e escrever na linha de comandos:

> *game-of-life*
- Seleccionar a configuração de células inicial (por ex., *glider*) e pressionar o botão *Start* para ver a evolução ao longo das gerações

Jogo da vida



Documentação do *Python*

- Digital
 - *Python web site*: www.python.org
 - *Python tutorials*: realpython.com/
 - *Python Package Index*: pypi.org
 - Comunidades, fóruns, *you tube*, etc.
- Papel
 - Livro: *Introduction to Computation and Programming Using Python*, de John V. Guttag (The MIT Press)
 - Revistas (com re-edições periódicas): *Coding Made Simple* (Future PLC) e *The Python Manual* (Black Dog Media Limited)

Fim da sessão 6

