

Fundamentos de Programação

1º Semestre 3º Ano

Python

Computador

→ O computador é uma máquina cuja função é manipular símbolos: são automáticos, universais, eletrônicos e digitais;

- * Uma vez alimentado c/ a informação necessária, trabalha por si só, sem a intervenção humana
- ↳ Recebe um programa → conjunto de instruções quanto ao modo de resolver o problema
 - escritas numa notação compreendida pelo computador
 - Especificam exatamente como o trabalho deve ser executado

- * Pode efectuar qualquer tarefa cuja solução possa ser expressa através de um programa

Operações Elementares

→ De acordo com a teoria de Church-Turing, qualquer computação pode ser baseada num pequeno número de operações elementares:

1. Operações de entrada de dados;
2. Operações de Saída de dados;
3. Operações matemáticas;
4. Execução condicional;
5. Repetição

→ A programação corresponde a dividir um programa grande e complexo em vários problemas, até podermos ser expressos por operações elementares.

* Efetua operações sobre informação que é codificada recorrendo a duas grandezas discretas (0 e 1) e não grandezas que variam de modo contínuo.

Algoritmos

→ Um programa é caracterizado matematicamente como um algoritmo, escrito numa linguagem que é entendida pelo computador

→ É uma sequência finita de instruções bem definidas e não ambíguas, cada uma das quais pode ser executada mecanicamente num período de tempo finito com uma quantidade de esforço finita.

* Existe uma ordem pela qual as instruções aparecem no algoritmo, e que estas instruções são em número finito;

* Significado de cada uma das instruções é claro, não existem múltiplas interpretações;

* A execução das instruções não requer imaginacão por parte dos executantes;

* A execução de cada uma das instruções termina.

→ Um algoritmo está sempre associado à resolução de um dado problema; a execução das instruções garante que o seu objetivo é atingido.

Características de um Algoritmo

→ Sequências finitas de instruções que devem ser seguidas de modo a atingir um determinado objetivo;

→ Um algoritmo deve possuir três características: ser nítido, eficiente e ter a garantia de terminar;

1. Deve especificar exacta e rigorosamente o que deve ser feito, não havendo lugar para ambiguidade; Para evitar a ambiguidade inherentemente à linguagem natural criam-se novas linguagens para exprimir os algoritmos de um modo rigoroso.

2. Cada instrução do algoritmo deve ser suficientemente básica e bem compreendida de modo a poder ser executada num intervalo de tempo finito, com uma quantidade de esforço finito;

3. O algoritmo deve levar a uma situação em que o objetivo tenha sido atingido e não existam mais instruções para serem executadas;

Programas e Algoritmos

→ Um algoritmo escrito de modo a poder ser executado por um computador tem o nome de programa

→ As operações, e consequentemente os resultados, são variáveis, após o que efetua operações sobre essas variáveis, possivelmente atribuindo valores a novas variáveis e, finalmente, chega a um conjunto de valores que constituem o resultado do programa.

→ As ações devem ser suficientemente elementares para poderem ser executadas facilmente pelo agente que executa o algoritmo; Estas ações elementares podem ser divididas isto é, podem referir-se a um conjunto de ações mais simples a serem executadas numa sequência bem definida.

► Linguagens de Programação

→ Linguagem usada para escrever programas de computador;
→ Podem ser classificadas em linguagens máquina, "assembly" e linguagens de alto nível.
→ A linguagem máquina é a linguagem usada para comandar diretamente as operações do computador; são constituídas por uma sequência de símbolos discretos, correspondendo à existência ou à ausência de sinal e manipulam diretamente entidades dentro do computador → é difícil de usar e compreender por humanos e rara de computador para computador;

→ A linguagem "assembly" é semelhante à linguagem máquina, diferindo desta no sentido em que usa nomes simbólicos com significado para humanos;

→ As linguagens de alto nível aproximam-se das linguagens que os humanos usam e podem ser utilizadas em computadores diferentes.

→ Para que os computadores possam "entender" os programas escritos numa linguagem de alto nível, existem programas que traduzem as instruções de linguagens de alto nível em linguagem máquina, chamados processadores da linguagem → compilação

► Sintaxe e Semântica

→ A sintaxe é o conjunto de regras que definem as relações válidas entre os componentes da linguagem → nada diz em relação ao significado das frases da linguagem // como apenas se preocupa com o processo de combinação dos símbolos de uma linguagem pode ser facilmente formalizada.

- Notação BNF:

• Para designar um componente da linguagem escrevemo-lo entre parênteses:
"⟨" e "⟩"; Os nomes entre ⟨ ⟩ são designados por símbolos não terminais

→ Interpretação
(Python)

• O símbolo " | " (ou) representa possíveis alternativas

• O símbolo " ::= " serve para definir componentes da linguagem

esse símbolo + imediatamente a seguir a um símbolo não terminal significa que esse símbolo pode ser repetido uma ou mais vezes.

• " * " significa que esse símbolo pode ser repetido zero ou mais vezes;

• " { } " significa que os símbolos entre elas são opcionais

- exemplo:

⟨ número binário ⟩ ::= < digito binário > | < digito binário > ⟨ número binário >

⟨ digito binário ⟩ ::= 0 | 1

→ PODE ORIGINAR RESULTADOS como:

0	01	10110	etc ...
1	011	01111011	

→ Nas linguagens de programação, a ambiguidade sintética não pode existir; A notação utilizada para definir formalmente uma linguagem, no caso da notação BNF, ⟨ ⟩, | , ::=, { }, +, *, os símbolos não terminais e os símbolos terminais, é denominada metalinguagem, visto ser a linguagem que utilizamos para falar acerca de outra linguagem;

→ A semântica de uma linguagem define qual o significado de cada frase da linguagem; cada frase em Python tem uma semântica, a qual corresponde às ações tomadas pelo Python ao executar essa frase. Esta semântica é definida por regras para extrair o significado de cada frase.

► Tipos de Erros num Programa

→ Podem ser de natureza sintática ou semântica;

* Sintáticos - resultam do facto de o programador não ter escrito as frases do seu programa de acordo com as regras da gramática da linguagem de programação utilizada; A deteção é feita pelo processador da linguagem, o qual fornece normalmente um diagnóstico sobre o que provavelmente está errado;

* Semânticos - são erros, em geral, muito mais difíceis de detectar; Resultam do facto do programador não ter expressado corretamente, através da linguagem de programação, as ações a serem executadas; podem manifestar-se pela geração de uma mensagem de erro

facilos que nunca terminam:

* Ao processo de deteção e correcção, dá-se o nome de depuração → "debugging" / e os erros chama-se "bugs" // Para desenvolver programas são necessárias duas competências fundamentais, a capacidade de resolução de problemas e a capacidade de depuração

o Expressões

- É uma entidade computacional que tem um valor;
 - Pode ser uma constante, uma expressão composta, um nome ou uma aplicação de função
- Em BNF, uma expressão é definida do seguinte modo:

$\langle \text{expressão} \rangle ::= \langle \text{constante} \rangle \mid \langle \text{expressão composta} \rangle \mid \langle \text{nome} \rangle \mid \langle \text{aplicação de função} \rangle$

o Constantes

- Números, valores lógicos ou cadeias de caracteres;
- O valor de uma constante é a própria constante;
- A representação externa de uma entidade corresponde ao modo como nós visualizamos essa entidade, independentemente do modo como esta é representada internamente no computador;

R. externa

1958

R. interna

10010100110

A Tipos de Constantes:

1. Números inteiros (não tem parte decimal)
2. Números reais (não com parte decimal)
3. Valores Lógicos (True e False)
4. Cadeias de caracteres (sequência de caracteres), aparecem delimitadas por aspas " " ; o seu comprimento é o número de caracteres do seu conteúdo.

o Expressões Compósitas

- Existem operações embutidas / pré - definidas / primitivas;
- As operações embutidas podem ser utilizadas através do conceito de expressão composta; corresponde ao conceito de aplicação de uma operação a operandos; é constituída por um operador e um certo número de operandos; os operadores podem ser unários (se têm apenas 1 operando) ou binários (se têm dois operandos ex: + *).
- Orden de aplicação de operadores numa expressão:
 1. Lista de prioridades dos operadores;
 2. Orden de aplic. dos op.

Prioridade
máxima funções
not, - (simétrico)
*, /, //, %
+ , - (subtração)
<, >, =, <=, >=, !=
and
OR

→ Da esquerda p/ a direita

o Tipos Elementares de Informação

→ Um tipo de informação é caracterizado por um conjunto de entidades (valores) e um conjunto de operações aplicáveis a essas entidades. Ao conjunto de entidades dá-se o nome de domínio do tipo. Cada uma das entidades do domínio do tipo é designada por elemento de tipo.

→ Dois Grupos: Inteiro, Real e Lógico

1. Elementares - as suas constantes (elementos) serem tomadas como indecomponíveis, por exemplo, "Vendadeiro" e "Falso" (tipo lógico);

2. Estruturados - as suas constantes são constituídas por um agregado de valores;

o Tipo Inteiro

→ Os números inteiros, designados por int, são números sem parte decimal, podem ser positivos, negativos ou zero;

Operação	Valor
$e_2 + e_2$	Soma
$e_1 - e_2$	Subtração
- e	Simétrico
$e_1 * e_2$	Multiplicação
$e_1 // e_2$	Divisão inteira
$e_1 \% e_2$	Resto da divisão inteira
$\text{abs}(e)$	Valor absoluto

O Tipo Real

Operação	Valor	Operação	Tipo Arg	Tipo valor	Operação
$e_1 + e_2$	soma	Round(e)	Real	int	Inteiro mais próximo do real e
$e_1 - e_2$	subtração	int(e)	Real	int	Parte inteira do real
$-e$	simétrico	float(e)	int	Real	Número real correspondente a e
$e_1 * e_2$	produto				
e_1 / e_2	divisão				
abs(e)	valor absoluto				

> Tipo Lógico

→ Designam-se por bool, pode assumir dois valores: True ou False;

→ As operações podem ser:

unárias

ou

binárias

~ Produzem um valor lógico a partir de um valor lógico. Apenas existe o not, que muda o valor lógico;

~ Aceitam dois argumentos do tipo lógico e produzem um valor do tipo lógico; Existe a conjunção e a disjunção;

↳ and
os dois são True
↓
True

↳ or
os dois são False
↓
False

> Nomes e Atribuição

→ A associação entre um nome e um valor é realizada através da instrução de atribuição, que recorre à operação encaixada =, o operador de atribuição;

Operação	Variável	Valor
$e_1 == e_2$	v	igual
$e_1 != e_2$	d	diferente
$e_1 > e_2$	m	maior
$e_1 \geq e_2$	mo	maior ou igual
$e_1 < e_2$	m	menor
$e_1 \leq e_2$	me	menor ou igual

> Predicados e Condições

→ Predicado: operação que produz resultados do tipo lógico;

→ Condição: expressão cujo valor é do tipo lógico;

> Leitura de Dados

→ input('informação')

>>> input('->')

→ 5

* Inferido através do teclado

'5'

carácter Escape

significado

\"	Barrar ao contrário (\`)
\'	Rica ('')
\"	Aspas
\a	Toque de Campainha
\b	Retrocesso de um espaço
\f	Salto de página
\n	Salto de linha
\r	"Return"
\t	Tabulação horizontal
\v	Tabulação vertical

→ eval(characters) → devolve um número/avalia

>>> eval('5')

5

```
>>> a = 30
>>> b = 15
>>> print('a=', a, 'b =', b)
a= 30 b = 15
```

```
>>> print('a=', a, '\nb =', b)
a= 30
b = 15
```

> Escrita de Dados

→ print(info)

=>

ESQUEMOS, INSTRUÇÕES E SEQUENCIAS

- No nível de linguagem máquina, existem 2 tipos de estruturas de controle: a sequência e o salto:
 - * As instruções são executadas pela ordem em que aparecem no programa;
 - * Transferência da execução para qualquer ponto do programa;
- Em linguagens de alto nível para além destas estruturas existem outras, como a iteração e a Repetição.

BNF

$\langle \text{instruções} \rangle ::= \langle \text{instrução} \rangle \boxed{\text{CR}} \mid$
 $\langle \text{instrução} \rangle \boxed{\text{CR}} \langle \text{instruções} \rangle$

$\boxed{\text{CR}}$ → corresponde ao fim da linha

Python

```
nota_1 = 17
nota_2 = 18
media = (nota_1 + nota_2) / 2
print ('A média é', media)
```

↳ A média é 17,5

Seleção

- Execução condicional de instruções

→ if → permite a seleção entre duas ou mais alternativas
 $>>> \text{if nota} > 5$!atenção à tabulação!

→ else → se não cumpre o if
 $>>> \text{nota} = 15$
 $>>> \text{if nota} > 16$
 $\quad \quad \quad \text{Print ('Bom trabalho')}$
else:
 $\quad \quad \quad \text{Print ('Podes fazer melhor')}$
Podes fazer melhor

→ elif → else com uma condição

Repetição

- Repetição da execução de um grupo de instruções ou de todo o programa;
- Ciclo finito
- while → enquanto uma expressão do tipo lógico for verdadeira
- Instrução break → quebra o ciclo

Soma = 0
num = eval(input('Escreva um inteiro\n(-1 para terminar)'))

while num != -1:

soma = soma + num

num = eval(input('Escreva um inteiro\n(-1 p/ terminar)'))

Print ('A soma é:', soma)

A quantidade de vezes que é preciso repetir o ciclo é desconhecida;

Funções

- Definir as funções por compreensão: chegar ao resultado final através dos valores de entradas scalars variáveis e a indicação de um processo de cálculo ($f(x) = x * x$);
- 1. A definição de uma função é feita fornecendo um nome para a função, uma indicação seu domínio ($f(s)$).
- 2. A aplicação da função é feita fornecendo o nome da função e um elemento do

def quadrado (x):

Return x * x → corpo da função

>>> quadrado (7)

49

→ Return → apenas dentro de funções / termina a execução do corpo da função;

Abstracção Procedimental

- Consiste em dar um nome à sequência de ações que servem para atingir um objetivo, e utilizando esse nome cada vez que desejamos atingir esse objetivo sem ter de especificar de novo todas as ações.
- Permite de desenvolver programas complexos, sem nos perdermos nos detalhes.

Erros

Nome

Situação

AttributeError Atributo não existente num objeto

ImportError Biblioteca não existente

IndexError Índice fora da gama de tuplo/lista

KeyError Chave inexistente

Eval/Input → enc. exp. sintaxe incorreta

Nome

Situação

NameError Nome que não existe

ValueError Argumento incorreto

ZeroDivisionError Divisão p/ zero

Raise: força a geração de um erro

Estrutura de bloco																							
→ Estrutura de blocos: Definir funções dentro das quais existem outras funções → um bloco pode ser considerada um bloco!																							
• Nóculos																							
→ Coleção de funções agrupadas num único ficheiro																							
<pre>>>> import math → módulo >>> math.pi → função >>> math.sqrt(4) 2.0 >>> math.sin(math.pi/2) 1.0</pre>	<p style="text-align: center;"><u>Python</u></p> <table> <tbody> <tr><td>pi</td><td>π</td></tr> <tr><td>e</td><td>e</td></tr> <tr><td>sin(x)</td><td>$\sin(x)$</td></tr> <tr><td>cos(x)</td><td>$\cos(x)$</td></tr> <tr><td>tan(x)</td><td>$\tan(x)$</td></tr> <tr><td>log(x)</td><td>$\ln(x)$</td></tr> <tr><td>exp(x)</td><td>e^x</td></tr> <tr><td>pow(x,y)</td><td>x^y</td></tr> <tr><td>sqrt(x)</td><td>\sqrt{x}</td></tr> <tr><td>ceil(x)</td><td>Maior inteiro $\geq x$</td></tr> <tr><td>floor(x)</td><td>Maior inteiro $\leq x$</td></tr> </tbody> </table>	pi	π	e	e	sin(x)	$\sin(x)$	cos(x)	$\cos(x)$	tan(x)	$\tan(x)$	log(x)	$\ln(x)$	exp(x)	e^x	pow(x,y)	x^y	sqrt(x)	\sqrt{x}	ceil(x)	Maior inteiro $\geq x$	floor(x)	Maior inteiro $\leq x$
pi	π																						
e	e																						
sin(x)	$\sin(x)$																						
cos(x)	$\cos(x)$																						
tan(x)	$\tan(x)$																						
log(x)	$\ln(x)$																						
exp(x)	e^x																						
pow(x,y)	x^y																						
sqrt(x)	\sqrt{x}																						
ceil(x)	Maior inteiro $\geq x$																						
floor(x)	Maior inteiro $\leq x$																						

Tuplos	
→ Frequência de elementos	
→ Referimo-nos aos elementos através da posição que ocupam: <u>índices</u>	-1 é o último elemento
→ Representação: (), (1,2,3), (1, True), (1,), (1, (1,1), 1)	consegam no 0
<pre>>>> notas = (15, 6, 10, 12, 12) >>> notas (15, 6, 10, 12, 12) >>> notas[0] 15 >>> notas[-2] 12 >>> i=1 >>> notas[i+1] 10</pre>	<pre>>>> a = ((1,2,3), 4, (5,6)) >>> a[0] (1,2,3) >>> a[0][1] 2</pre>
SÃO IMUTÁVEIS	
Operações	
 <ul style="list-style-type: none"> t₁ + t₂ t * i t[i₁:i₂] e int e not int tuple(a) len(t) 	<u>Valores</u> <ul style="list-style-type: none"> concatenação Repetição i vezes do tuplo sub-tuplo de t entre i₁ e i₂-1 True se e pertence a t True se e ñ pertence a t Transforma o seu arg num tuplo Número de elementos do tuplo

arg	→ isinstance: tem orador True apenas se o tipo da expressão do 1º arg corresponde ao seu 2º
	>>> isinstance(3,int) True
• Códigos Contados	
→ for	→ range → geração de sequência de elementos
>>> for i in (1,3,5): print(i)	<ol style="list-style-type: none"> 1. range(ex) devolve a sequência contendo os inteiros entre 0 e e₁-1 / ex: range(2) = (2,2) 2. range(e₁, e₂) devolve a sequência entre e₁ e e₂-1; ex: range(-3,3) = (-3,-2,-1,0,1,2) 3. range(e₁, e₂, e₃) devolve sequência começa em e₁, nunca supera e₂-1, em que cada elemento é obtido pela soma do elemento anterior com e₃ / ex: range(2,20,3) = (2, 5, 8, 11, 14, 17)

Códigos de Caracteres Revistados	
→ str	
→ help(<nome>) mostra a <u>definição</u> associada a <nome> ~ delimitada por "''' def """	
 Mesmas operações	
→ ord(a) → recebe char e devolve o código decimal que o representa	
→ chr(a) → recebe um nº positivo e devolve o carácter	unicode
>>> ord('R')	>>> chr(125)
82	'ñ'

Listas	
→ list: é uma sequência de elementos	
Operação	Valor
l ₁ + l ₂	concatenação
l * i	Repetição i x da lista
l[i ₁ :i ₂]	sub-lista entre i ₁ e i ₂ -1
del(els)	Remove os elementos especificados
Operação	Valor
e in l	True se e pertence à lista
e not in l	True se e ñ pertence à lista
list(a)	Transforma o arg em lista
len(l)	Derive o nº de elementos

seus mutáveis, logo $lst = [1, 2, 3, 4]$
 >>> $lst[2] = 'a'$
 print lst
 [1, 2, 'a', 4]

Métodos de Passagem de Parâmetros

→ Passagem por valor: A função Recebe o valor de cada um dos parâmetros e nenhuma informação adicional; exemplo:

```
def troca(x,y):
    x,y = y,x # os valores são trocados
>>> x = 3 } criação de x e y no ambiente global
>>> y = 10 } local
>>> troca(x,y) } local
>>> x
3
>>> y
10
```

UNIDIRECIONAL

→ Quando a função troca é executada, o ambiente local desaparece, voltando-se aos valores de x e y iniciais.

→ Passagem por Referência: Os parâmetros formais e concretos partilham o mesmo ambiente:

```
def troca_2(lst, i1, i2):
    lst[i1], lst[i2] = lst[i2], lst[i1]
>>> lista = [0, 1, 2, 3, 4, 5]
>>> troca_2(lista, 2, 5)
>>> lista
[0, 1, 5, 3, 4, 2]
```

Algoritmos de Procura

→ Procura sequencial: Começar no primeiro elemento da lista e comparar sucessivamente o elemento procurado com o elemento na lista:

```
def procura(lst, chave):
    for i in range(len(lst)):
        if lst[i] == chave:
            return i
    return -1
```

→ Procura Binária: Lista ordenada → considera-se o elemento do meio da lista x, se x > procura, está na 1ª metade → Repetir o processo p/ a 2ª metade.

```
def procura(lst, chave):
    linf = 0
    lsup = len(lst) - 1
    while linf <= lsup:
        meio = (linf + lsup) // 2
        if chave == lst[meio]:
            return meio
        elif chave > lst[meio]:
            lins = meio + 1
        else:
            lsup = meio - 1
    return -1
```

Algoritmos de Ordenação

→ Ordenação

interna

• Ordenam um conjunto de elementos que estão simultaneamente armazenados em memória (ex: lista)

→ Ordenação por bolhamento

• Percorrer os elementos a ordenar, comparando elementos adjacentes, trocando pares de elementos que se encontram fora de ordem;

→ Ordenação Shell: comparar e trocar elementos separados por um certo intervalo; esse intervalo é dividido ao meio e o processo repete-se com o novo intervalo

→ Ordenação por Seleção: em cada passagem colocar um elemento na sua posição correta;

Funções Recursivas

→ Função definida em termos de si própria, exemplo:

• Recursiva def factorial(n):

```
fact = 1
for i in range(n, 0, -1):
    fact = fact * i
return fact
```

• Recursiva: def factorial(n):

```
if n == 0:
    return 1
else:
    return n * factorial(n-1)
```

→ Encadeamento:

```

f(3)
3 * f(2)
3 * (2 * f(1))
3 * (2 * (1 * f(0)))
3 * (2 * (1 * 1))
3 * (2 * 1)
3 * 2
6

```

1. Caso terminal: versão mais simples do problema para a qual a solução é conhecida;

2. Caso geral: o problema é definido em termos de uma versão mais simples de si próprio;

• Funções de Ordem Superior

→ funções como Parâmetros:

```

def somatorio (calc - termo, linf, prox, lsup):
    soma = 0
    while linf <= lsup:
        soma = soma + calc - termo (linf)
        linf = prox (linf)
    return soma

>>> somatorio (identidade, 1, inc1, u0)
    ↳ utilizam-se funções como parâmetros

```

lambda x: x+1, devolve o valor do seu argumento mais um

```
>>> somatorio (lambda x: x, 1, lambda x: x+1, u0)
```

lambda x: 1 / quadrado (x) = inv-quadrado (x)

```

def factorial (n):
    if n == 0:
        return 1
    else:
        return n * factorial (n-1)

```

• Funcionais sobre Listas

→ Um transformador é um funcional que recebe como argumentos uma lista e uma operação aplicável aos elementos da lista, e devolve uma lista em que cada elemento resulta da aplicação da operação ao elemento correspondente da lista original:

```

def transforma (tr, lista):
    res = list ()
    for e in lista:
        res = res + [tr (e)]
    return res

```

→ Um filtro é um funcional que recebe como argumentos uma lista e um predicado aplicável aos elementos da lista, e devolve a lista constituída apenas pelos elementos que satisfazem o predicado:

```

def filtra (teste, lista):
    res = list ()
    for e in lista:
        if teste (e):
            res = res + [e]
    return res

```

→ Um acumulador recebe como argumentos uma lista e uma operação aplicável aos elementos da lista, e aplica sucessivamente essa operação aos elementos da lista, devolvendo o resultado da aplicação da operação a todos os elementos da lista:

```

def acumula (fn, lst):
    res = lst [0]
    for i in range (1, len (lst)):
        res = fn (res, lst [i])
    return res

```

• Funções como Valores ou Funções

→ ex: def derivada(x): def quadrado(y): >>> derivada (quadrado)(3)

valor

valor = derivada do ponto 3²

- Funções Recursivas
- O conceito de repetição é realizado exclusivamente através da recursão;
 - exemplo: $x^n = \begin{cases} 1 & \text{se } n=0 \\ x \times (x^{n-1}) & \text{se } n>1 \end{cases}$
- def potencia(x, n):
- ```

if n == 0:
 Return 1
else:
 Return x * potencia(x, n-1)

```
- Recurssão Linear
- Gera-se um encadeamento de operações suspensas à espera do valor de outras expressões:
    - Operação sucessivamente adiada (fase de expansão)
    - Execução das operações (fase de contração)
  - A um processo recursivo que cresce linearmente com o valor dâ-se o nome de processo recursivo linear;
- Iteração Linear
- def factorial-aux (prod-ac, prox):
- ```

if prox == 0:
    Return prod-ac
else:
    Return factorial-aux (prod-ac * prox, prox - 1)

```
- def factorial(n):
- ```

def factorial-aux (prod-ac, prox):
 ...
 Return factorial-aux (1, n)

```
- Não se gera um encadeamento de funções suspensas / tem variáveis de estado e uma regras que especifica como atualizá-las
- Recurssão com Processos e Funções
- A evolução de processos pode ser classificada como uma evolução recursiva ou como uma evolução iterativa:
    - Um processo recursivo tem uma fase de expansão seguida de uma de contração. O computador mantém informação "escondida" que regista o ponto onde está o processo na cadeia de operações adiadas;
    - Um processo iterativo não cresce nem contrai
- Recurssão em Árvore
- Múltiplas faces de crescimento e contração → ineficiente
- Recurssão de Cadeia
- Primeiro o cálculo é realizado, depois a chamada recursiva;
  - A chamada recursiva é a última operação realizada pela função, não existindo operações adiadas;
- ```

def factorial(n, acc):
    if n == 0:
        Return acc
    else:
        Return factorial (n-1, n * acc)

```
- fica
- ```

def factorial (n):
 def factorial-aux (n, acc):
 if n == 0:
 Return acc
 else:
 Return factorial-aux (n-1, n * acc)
 Return factorial-aux (n, 1)

```
- O Tipo Ficheiro
- Localizar e utilizar: ler ou escrever;
  - Open (abre o ficheiro)
  - 'R', 'w', 'a' aberto / escrita a partir do inicio
- aberto p/ escrita a partir do final
- FP

→ encoding = <tipo> → expressão que representa o tipo de codificação de caracteres utilizada no ficheiro

## Leitura de Ficheiros

t = open ('teste.txt', 'R', encoding = 'UTF-16')

### Operações:

1. t.readline(): lê a linha do ficheiro que se encontra imediatamente a seguir ao indicador de leitura. Se o indicador de leitura estiver no fim do ficheiro, esta função tem o valor ''.

2. t.readlines(): lê todos os caracteres do ficheiro que se encontram depois do indicador de leitura, tendo como valor uma lista em que cada um dos elementos é a cadeia de caracteres correspondente a cada uma das linhas que foi lida;

3. t.read(): lê todos os caracteres do ficheiro, devolve cadeia de caracteres;

4. t.close(): fecha o ficheiro;

```
>>> f = open('teste.txt', 'R', encoding = 'UTF-16')
```

```
>>> l1 = f.readline()
```

```
>>> l1
```

'Este é um teste \n'

```
>>> l2 = f.read()
```

```
>>> l2
```

' que mostra como o Python lê ficheiros '

```
>>> print(l2)
```

que mostra como o Python  
lê ficheiros

```
>>> g = open('exemplo.txt', 'R', encoding = 'UTF-16')
```

```
>>> lines = g.readlines()
```

['Este é um teste \n', ' que mostra como o Python \n', ' lê ficheiros de caracteres \n']

## Escrita em ficheiros

### Operações:

1. t.write(chars): escreve chars em t. Devolve o nº de caracteres;

2. t.writelines(lines): seq = lista ou tuplos cujos elementos são chars, não escreve o carácter de fim de linha; Não devolve valor;

3. t.close(): fecha o ficheiro;

→ Se um ficheiro é aberto para escrita, o seu conteúdo é apagado com a operação de abertura do ficheiro.

## O tipo Dicionário

→ dict

→ Tipo mutável, contém um conjunto de pares

↓  
1º elemento  
↓ chave

↓  
2º elemento  
↓ valor associado à chave

→ São representados dentro de chaves {}  
→ A chave tem de ser de um tipo imutável;  
exemplos:

```
d = {'a': 3, 'b': 2, 'c': 4} d['c'] = 4
```

```
>>> ex-dic['a'] = ex-dic['a'] + 1
```

```
>>> ex-dic['a']
```

| Operação   | Valor                                    |
|------------|------------------------------------------|
| c in d     | True se a chave c pertence ao dicionário |
| c not in d | True se a chave c não pertence a d       |
| len(d)     | Nº de elementos de d                     |

```
n = {'a': 1, 'b': 2, 'c': 3}
```

```
>>> for i in n:
 print(i, n[i])
```

a 1

b 2

c 3

## Ler um ficheiro de texto

```

fis = open('Lusadas.txt', 'r', encoding='UTF-16')
cont = {}
linha = fis.readline()
while linha != '':
 for c in linha:
 if c not in cont:
 cont[c] = 1
 else:
 cont[c] = cont[c] + 1
 linha = fis.readline()
fis.close()

```

>>> ex\_cadeia = 'Exemplo DE CONVERSÃO'

>>> ex\_cadeia.lower()

'exemplo de conversão'

## Dicionários de Diccionários

→ {'FP': {'2010/11': 12, '2011/12': 15}, 'AL': {'2010/11': 10}, 'TC': {'2010/11': 12},  
 'SD': {'2010/11': 'REP', '2011/12': 13}}

## A Abstracção em Programação

→ A abstracção de dados é uma metodologia que permite separar o modo como estrutura de informação é utilizada dos pormenores relacionados com o modo como essa estrutura de informação é construída a partir de outras estruturas de informação

## Tipos Abstratos de Informação

→ Separação das partes dos programas que lidam com o modo como as entidades do tipo são utilizadas das partes que lidam com o modo como as entidades são representadas.

1. Identificação das operações básicas
2. Axiomatização das operações básicas
3. Escolha de uma representação
4. Concretização das op. básicas

## Identificação das Operações Básicas

- Construtores: construir novos elementos do tipo;
- Selectores: accedem aos constituintes dos elementos do tipo;
- Modificadores: alteram destrutivamente os elementos do tipo, ex.: del i
- Transformadores: transformam os elementos de um tipo para outro tipo;
- Reconhecedores: identificam elementos do tipo;
- Testes: comparações entre os elementos do tipo;

## Objetos

→ Entidade computacional que apresenta informação interna e um conjunto de operações que definem o seu comportamento;

1. A palavra class indica que se trata da definição de um objeto; pode apenas ser definido um nome; pode ser fornecido um nome seguido de outro nome entre () e representa o nome do objeto
2. Definição de um método que está associado ao objeto; existe sempre um cujo
3. Os parâmetros formais de um método contêm sempre, como primeiro elemento, um parâmetro com o nome self.

```

class complex:
 def __init__(self, real, imag):
 if isinstance(real, (int, float)) and isinstance(imag, (int, float)):
 self.r = real
 self.i = imag
 else:
 raise ValueError('complex: argumento inválido')

 def p_real(self):
 return self.r

```

```

def p-imag(self):
 Return self.i

def compl-zero(self):
 Return self.r == 0 and self.i == 0

def imag-puro(self):
 Return self.r == 0

def compl-iguais(self, outro):
 Return self.r == outro.p-real() and self.i == outro.p-imag()

def escrever(self):
 if self.i >= 0:
 print(f'{self.r} + {self.i}i')
 else:
 print(f'{self.r} - {abs(self.i)}i')

```

→ c1.p-Real é o nome do método que deriva a parte real de c1

```

>>> c1 = compl(3, 2)
>>> c1.p-Real()
3
>>> c1.p-imag()
2
>>> c1.escrever()
3+2i
>>> c2 = compl(2, -2)
>>> c2.escrever()
2-2i

```

```

>>> c1 = compl(3, 5)
>>> type(c1)
<class '-main-.compl'>
>>> c1 = compl(9, 6)
>>> isinstance(c1, compl)
True

```

→ Uma classe corresponde a uma infinidade de objetos com as mesmas variáveis e com o mesmo comportamento.

### • O Tipo Conta Bancária

→ class conta:

```

def __init__(self, quantia):
 self.saldo = quantia

def consulta(self):
 Return self.saldo

def deposito(self, quantia):
 self.saldo = self.saldo + quantia
 Return self.saldo

def levantamento(self, quantia):
 if self.saldo - quantia >= 0:
 self.saldo = self.saldo - quantia
 Return self.saldo
 else:
 print('Saldo insuficiente')

```

```

class
* conta-gen

```

" "

" "

" "

```

def levantamento(self, quantia):
 self.saldo = self.saldo - quantia
 Return self.saldo

```

### • Classe, Subclasse e Herança

→ Uma classe é uma subclasse de outra classe, se a primeira corresponder a uma especialização da segunda; O comportamento da subclasse corresponde ao comportamento da superclasse, exceto no caso em que o comportamento específico está indicado para a subclasse; A subclasse herda o comportamento da superclasse;

class conta-ordenado(conta-gen):

```

def __init__(self, quantia, ordenado):
 if quantia >= ordenado:
 self.saldo = quantia
 self.ordenado = ordenado
 else:
 print('O saldo deve ser maior que o ordenado')

def levantamento(self, quantia):
 if quantia <= self.saldo - self.ordenado:
 self.saldo = self.saldo - quantia
 Return self.saldo
 else:
 print('Saldo insuficiente')

```

Exercício

→ Uma operação é polimórfica quando é possível definir funções diferentes que usam a mesma operação para lidar com tipos de dados diferentes;

```

>>> a = 5
>>> a.___add__ (2)
 +
-- sub -- - eq -- ==
-- mul -- * __repr__ -- → char
-- truediv -- /

```

class compl:

```

def __init__(self, real, imag):
 if isinstance(Real, (int, float)) and isinstance(Imag, (int, float)):
 self.R = Real
 self.i = Imag
 else:
 raise ValueError('complexo: arg errados')
def p-real(self):
 return self.R
def p-imag(self):
 return self.i
def compl_zero(self):
 return self.R == 0 and self.i == 0
def __eq__(self, outro):
 return self.R == outro.preal() and self.i == outro.p-imag()
def __add__(self, outro):
 R = self.p-real() + outro.p-real()
 i = self.p-imag() + outro.p-imag()
 return compl(R, i)

```

R + é o add

Pilhas

→ Estruturas de informação constituídas por uma sequência de elementos, são retirados pela ordem inversa pela qual foram colocados, põe-se no topo, tira-se do topo; Não se pode aceder ou inspecionar nenhum elemento exceto o que está no topo

Operações Básicas para Pilhas IMUTÁVEIS / MUTÁVEIS

→ Como Imutáveis:

1. Construtores: constroem pilhas → cria pilha e insere elemento (no topo) nova-pilha
2. Seletores: selecionam partes → indica o topo e remove topo tira
3. Reconhecedores: identificam tipos de pilhas → tipo e vazia pilha-vazia
4. Testes: iguais pilhas-iguais

→ Como Mutáveis:

1. Construtores: nova-pilha
2. Modificadores: empurra (+ el no topo) e tira (remove o topo)
3. Seletores: topo (indica o el no topo)
4. Reconhecedores: = imutáveis
5. Testes: = imutáveis

→ A pilha com elementos 357 (em que 3 está no topo) é representada:

3  
 5  
 7  
 = = =

Ariometria

→ Devem verificar-se as seguintes relações:

pilha(nova-pilha()) = verdadeiro

pilha(empurra(p, e)) = verdadeiro

pilha(tira(p)) = { verdadeiro se p não vazia  
falso caso contrário}

pilha-vazia(nova-pilha()) = verdadeiro

pilha-vazia(empurra(p, e)) = falso

topo(empurra(p, e)) = e

tira(empurra(p, e)) = p

pilhas-iguais(p1, p2) = { pilha-vazia(p1) se pilha-vazia(p1)  
pilha-vazia(p1) se pilha-vazia(p2)  
topo(p1) = topo(p2) se  
tira(p1) = tira(p2)  
falso caso contrário}

## O Representação de Pilhas

→ Recorrendo a listas:

1. Pilha vazia é a lista vazia;
2. O primeiro elemento corresponde ao topo;

## A Realização de Operações Básicas

→ >>> teste = [1, 2]

>>> teste[2] > 0

IndexError: list index out of range

>>> try:

    teste[2] > 0

except IndexError:

    print('Enganou-se no índice')

Enganou-se no índice

def nova\_pilha():

    return []

def empurra(pilha, elementos):

    return [elemento] + pilha

def topo(pilha):

    try:

        return pilha[0]

    except IndexError:

        raise ValueError('A pilha não tem elementos')

def tira(pilha):

    try:

        return pilha[1:]

    except IndexError:

        raise ValueError('A pilha vte')

def pilha(x):

    if x == []:

        return True

    elif isinstance(x, list):

        return True

    else:

        return False

def pilha\_vazia(pilha):

    return pilha == []

def pilhas\_iguais(p1, p2):

    return p1 == p2

def mostra\_pilha(pilha):

    if pilha != []:

        for e in pilha:

            print(e)

        print('==')

..

class pilha:

    def \_\_init\_\_(self):

        self.p = []

    def empurra(self, elemento):

        self.p = [elemento] + self.p

    return self.

    def tira(self):

        try:

            del (self.p[0])

        return self

    except IndexError:

        print('APNTE')

```

 def __str__(self):
 self.p == []
 -- Rep -- (self):
 if self.p != []:
 Rep = ''
 for e in self.p:
 Rep = Rep + ' ' + str(e) + '\n'
 Rep = Rep + '==='
 return Rep
 else:
 return '=='

```

### Filas

→ Sequência de elementos retinados pela ordem em que foram colocados; Apenas o 1º elemento pode ser adicionado ou removido;

### Operações Básicas para filas

- 1. Construtores: nova-fila
- 2. Seletores: inicio e fim
- 3. Modificadores: coloca (novos elem.) e retira
- 4. Transformadores: produz uma lista com os el da fila
- 5. Reconhecedores:
- 6. Testes: filas-iguais;

### A Classe Fila

```

class Fila:
 def __init__(self):
 self.f = []
 def inicio(self):
 try:
 return self.f[0]
 except IndexError:
 print('inicio: a fila não tem el')
 def comprimento(self):
 return len(self.f)
 def coloca(self, elemento):
 self.f = self.f + [elemento]
 return self.f
 def retira(self):
 try:
 del(self.f[0])
 return self.f
 except IndexError:
 print('... ')
 def fila_p_lista(self):
 lst = []
 for i in range(len(self.f)):
 lst = lst + [self.f[i]]
 return lst
 def fila_vazia(self):
 return self.f == []
 def filas_iguais(self, outra):
 outra_lista = outra.fila_p_lista()
 if len(self.f) != len(outra_lista):
 return False
 else:
 for i in range(len(self.f)):
 if self.f[i] != outra_lista[i]:
 return False
 return True

```

```

def __repr__(self):
 f = '< '
 if self.f != []:
 for i in range(len(self.f)):
 f += self.f[i].__repr__() + ','
 f += '> '
 return f

```

## • funções sobre listas

- Filtra, Transforma, Acumula
  - filter
  - map
  - Reduce

```

(lambda x,y: x+y)(1,2) : lambda x: x * factorial(x-1) if x>1 else 1
>>> 3 | x=4
 | >>> 24

```

map (lambda x: x\*\*3, range(5,10)) → lista com 5<sup>3</sup>, ..., 9<sup>3</sup>

Reduce (lambda x,y: x+y, [1,2,3]) → 6

filter (lambda x: x>50, [1,44,60]) → [60]

Maria Ribeiro  
30/01/2029