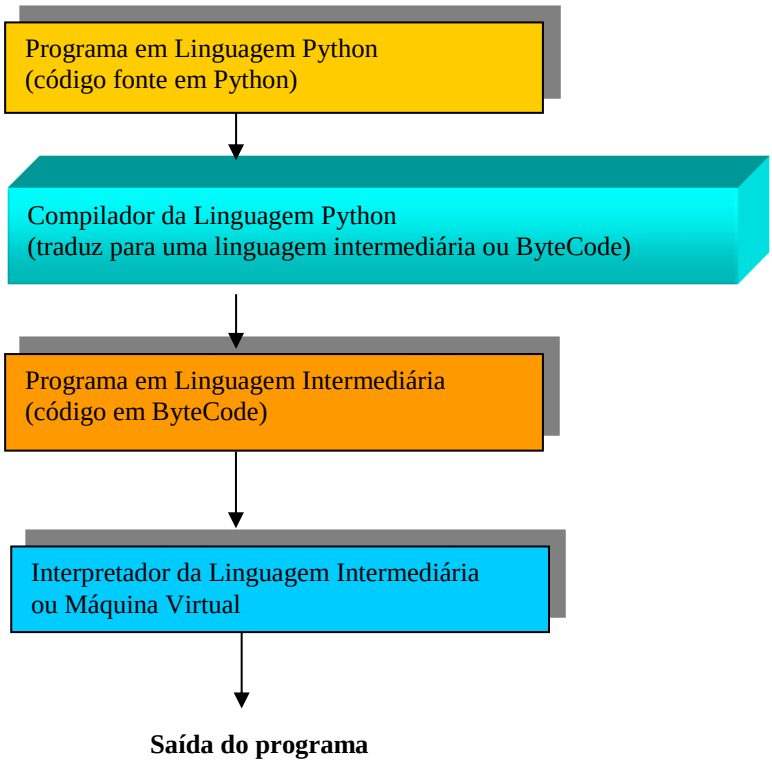
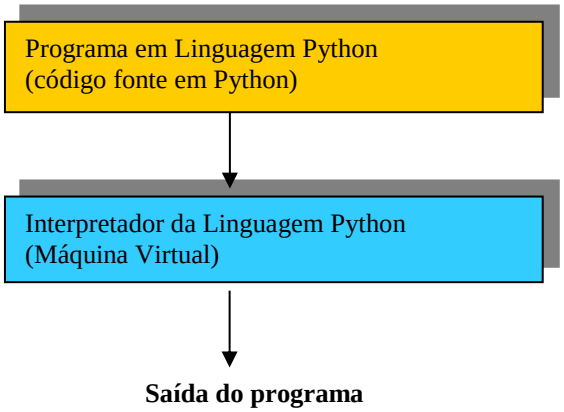
Python – Revisão e tópicos especiais

Python é uma linguagem interpretada, ou seja, o compilador Python traduz o programa fonte para uma linguagem intermediária (em geral chamada de ByteCode) que será interpretada por um programa interpretador também chamado de Máquina Virtual.

Um programa interpretado é menos eficiente que um programa compilado (traduzido diretamente para a linguagem de máquina), mas apresenta outras facilidades bastante interessantes como a portabilidade (roda em qualquer ambiente que tenha a Máquina Virtual). Além disso, as linguagens interpretadas permitem construções que facilitam sobremaneira a programação.



Podemos abstrair a existência da Linguagem Intermediária e pensar apenas que o programa em Python é simplesmente interpretado.



Ambiente de desenvolvimento Python

Existem vários ambientes de desenvolvimento para o Python (IDE – Integrated Development Environment). Em geral, estão integrados um editor de texto, o compilador e a Máquina Virtual. Permitem o desenvolvimento e teste completo do sistema. É o caso do ambiente IDLE (Integrated

DeveLopment Environment ou Integrated Development and Learning Environment) que usamos no curso.

Uso do Interpretador da Linguagem Python

O interpretador pode ser usado no modo *linha de comando* ("shell mode") e no modo de *script* ("program mode"). No modo de linha de comando, você digita comandos ou expressões em Python e o interpretador mostra o resultado se houver a cada linha digitada.

O exemplo a seguir ilustra o funcionamento no modo linha de comando:

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a = 32
>>> b = 15
>>> c = a + b
>>> c
47
>>> print(c)
47
>>> a + b
47
```

`>>>` O símbolo `>>>` é chamado de **prompt** do Python. O interpretador usa o prompt para indicar que está pronto para receber um comando. Ao digitar `2 + 3`, o interpretador avalia a expressão e responde `5`. A seguir, ele fornece um novo prompt na linha seguinte, indicando que ele está pronto para um novo comando.

O modo linha de comando é conveniente para testar pequenos pedaços de código, pois você recebe uma resposta rapidamente. Para qualquer coisa maior que algumas linhas é melhor usar o modo script.

No modo script podemos escrever um programa inteiro em um arquivo texto e usar o interpretador para executar o conteúdo do arquivo como um todo. Pode ser usado qualquer editor de texto, inclusive o do próprio interpretador, para gerar arquivo com extensão `.py` (exemplo: "`meu-programa.py`").

Python é uma linguagem orientada a objetos

A linguagem Python é orientada a objeto. Assim como C++, C♯, VB.NET, Java, Object Pascal, Objective-C, SuperCollider, Ruby e Smalltalk.
Os objetos são agrupados em classes pré-existentes na linguagem (intrínsecas ou built-in) ou que podem ser criadas por um programa.

As classes pré-existentes são:

Classe	Descrição	Objeto imutável?
int	número inteiro	sim
float	número em ponto flutuante	sim
bool	booleano	sim
listas	sequência de objetos	não
tuplas	sequência de objetos	sim
str	cadeia de caracteres	sim
set	sequência não ordenada de objetos	não
frozenset	sequência não ordenada de objetos	sim
dict	dicionário associativo	não

Identificadores ou variáveis

Fazendo uma analogia, os identificadores ou variáveis em Python são o mesmo que os ponteiros em C++ ou as referências em Java. Ou seja, um identificador contém um endereço de memória no qual está o objeto associado. Eventualmente o identificador pode estar associado ao objeto None (objeto vazio).

Identificadores ou variáveis podem ter qualquer combinação de letras, dígitos e underline, não podem iniciar com dígitos (`8coca_cola` por exemplo, é inválido) e não podem ser igual a uma das seguintes palavras reservadas do Python:

```
False      as         continue   else       from       in
not        return     yield      None       assert     def
except     global     is         or         try        True
break      del        finally    if         lambda     pass
while      and        class      elif       for        import
nonlocal   raise      with
```

Tipos dinâmicos

Tipos em Python são dinâmicos. O mesmo identificador pode indicar objetos de tipos diferentes durante o programa. Como no exemplo abaixo. A função type() retorna a classe do objeto.

```
# x é um int
x = 1
print(type(x))
# agora x é um str
x = "abcd"
print(type(x))
```

será impresso:

```
<class 'int'>
<class 'str'>
```

A construção de objetos

O processo de criação de um objeto de uma determinada classe é chamado de instanciação desta classe. Dizemos que criamos uma instância desta classe, ou a instância de um objeto desta classe.
A maneira mais simples de criar um novo objeto é o comando de atribuição.

identificador = construtor_da_classe(parâmetros)

Muitos das classes built-in não precisam do construtor. Assim: x = 8.6 cria um objeto do tipo float. É equivalente a x = float(8.6). O método float é o construtor_da_classe.

Outra forma de instanciar uma classe é chamar uma função que retorna um novo objeto. Abaixo, uma função que recebe uma lista de inteiros e devolve outra lista ordenada por valor.

```
x = [5, 3, 6, 2, 1]
y = ordena(x)
# y é um novo objeto. Supondo que a função ordena devolve
# uma nova lista com os mesmos elementos em outra ordem
```

Objetos mutáveis e imutáveis

A maioria dos objetos intrínsecos são imutáveis. Isso significa que não são alteráveis. Objetos das classes int, float, bool, tuplas, str e frozenset são imutáveis enquanto listas, set e dic não são.

Veja o que será impresso no exemplo abaixo. A função id() devolve o endereço de memória do objeto:

```
a = 5
b = a       # a e b referem-se ao mesmo objeto (5).
print(a, id(a), b, id(b))
a = a + 1   # um novo objeto é criado (6).
print(a, id(a), b, id(b))
c = 5           # c e b são o mesmo objeto
print(c, id(c))

x = [1, 2, 3]
y = x       # x e y referem-se ao mesmo objeto.
print(x, id(x), y, id(y))
x[1] = -1   # x e y referem-se ao objeto [1, -1, 3].
print(x, id(x), y, id(y))
```

Será impresso:

```
5 1655727264 5 1655727264
6 1655727280 5 1655727264
5 1655727264
[1, 2, 3] 58254320 [1, 2, 3] 58254320
[1, -1, 3] 58254320 [1, -1, 3] 58254320
```

Se o objeto for imutável, por exemplo, uma tupla, o comportamento de uma sequência como a anterior será diferente:

```
x = (1, 2, 3)
y = x       # x e y referem-se ao mesmo objeto.
print(x, id(x), y, id(y))
x[1] = -1   # neste comando teremos erro.
print(x, id(x), y, id(y))
```

Será impresso:

```
(1, 2, 3) 50917248 (1, 2, 3) 50917248
Traceback (most recent call last):
  File "C:/Users/msanches/Documents/Marcilio/Python - fontes -
Marcilio/teste id objetos - 1.py", line 17, in <module>
    x[1] = -1     # neste comando teremos erro.
TypeError: 'tuple' object does not support item assignment
```

O fato de listas serem objetos mutáveis nos permite fazer uma função que recebe uma lista e altera essa lista. Por exemplo, uma função que ordena uma lista trocando os elementos de lugar.

```
x = [5, 3, 6, 2, 1]
ordena(x)
```

De fato essa função já existe dentro da classe lista. É uma função built-in do Python:

```
x = [5, 3, 6, 2, 1]
x.sort()
```

A classe set

A classe set (conjuntos) permite a manipulação de sequências como se fossem conjuntos da álgebra. Um conjunto é representado com os seus elementos entre {}. Importante a distinção de sets e listas. Num set não há elementos repetidos e não há uma ordem dos elementos. Portanto, não se pode usar indexação como em listas. Outra restrição importante é que os elementos de um set só podem ser objetos imutáveis. Não podemos ter set de sets ou set de listas. Um frozenset é um set imutável.
Da mesma forma que em listas, podemos adicionar, retirar ou modificar elementos de um set mas não de um frozenset.

```
s1 = {1, 2, 3}
s2 = {2, 3, 4, 7}
print(s1 | s2) # união
print(s1 & s2) # intersecção
# pertinência
for i in range(5): print(i, i in s1)
```

Será impresso:

```
{1, 2, 3, 4, 7}
{2, 3}
0 False
1 True
2 True
3 True
4 False
```

O conjunto vazio não é representado por {} como esperado. O símbolo {} representa um dic vazio. Para gerar um conjunto vazio usamos o construtor da classe:

```
s1 = set() # s1 contém o conjunto vazio
```

O construtor da classe pode ser usado também para construir conjuntos a partir de listas ou strings.

```
sx = set([1, 3, 5])
sy = set('aeiou')
sz = frozenset([2, 3, 5, 7, 11])
print(sx, sy, sz)
```

Saída:

```
{1, 3, 5} {'o', 'i', 'u', 'a', 'e'} frozenset({2, 3, 5, 7, 11})
```

A classes dict

A classe dict (dictionare) permite associar elementos dentre de uma lista.

```
# associa strings a números
ds = {"branco":23, "preto":107, "vermelho":-234, "amarelo":31}
for st in ds: print(st)
for st in ds: print(ds[st])

# alterando e aumentando ds
ds["amarelo"] = -31
```

```
ds["azul"] = 1739       # com atribuição a novo elemento
ds.update({"preto":108, "violeta":-46}) # usando o método update
print(ds)
```

Será impresso:

```
branco
preto
vermelho
amarelo
23
107
-234
31
{'branco': 23, 'preto': 108, 'vermelho': -234, 'amarelo': -31, 'azul':
1739, 'violeta': -46}
```

Expressões

Operadores e operandos podem ser combinados de modo a formar as expressões aritméticas, lógicas e dos demais tipos de objetos. A semântica de uma expressão depende do tipo de operando. Por exemplo:

a + b tem um valor numérico (int ou float) se a e b forem numéricos
a + b é uma concatenação se a e b forem strings ou listas

Operadores lógicos : not – and - or

Quando uma expressão do tipo <oper-1> and <oper-2> ou <oper-1> or <oper-2> é calculada, eventualmente não é necessário avaliar o valor de <oper-2> (quando <oper-1> é False no and e <oper-1> é True no or).

Suponha um trecho que verifica se os elementos de uma lista de n elementos são crescentes. Quando i for n não calcula o valor de a[i-1] <= a[i]

```
i = 1
while i < n and a[i-1] <= a[i]: i = i + 1
```

Quando i for igual a n a parte da expressão a[i-1] <= a[i] não é avaliada, o que provocaria um erro de índice inválido.
Se modificarmos a ordem os operandos, a solução ficará errada, embora a operação and seja comutativa.

Operadores de igualdade: is, is not, == e !=

x is y é True quando x e y são o mesmo objeto, ou seja, x e y são sinônimos.
a == b é True quando os objetos referenciados por a e b são iguais.

x is not y é True quando x e y são objetos distintos, mesmo que tenham o mesmo valor.
a != b é True quando os objetos referenciados por a e b são diferentes.

Se x is y é True, então x == y.
Se x is not y é True, x == y ou x != y

Operadores de bits (bitwise)

São operadores para valores inteiros. Operam sobre os bits que representam os valores.

~ - complemento
& - and
| - or
^ - ou exclusivo

<< - deslocamento para a esquerda
>> - deslocamento para a direita

Veja o exemplo:

```
a = 1
b = 255

print(~a)
print(a & b)
print(a | b)
print(a ^ b)
print(a << 5)
print(b >> 5)
```

Será impresso:

```
-2
1
255
254
32
7
```

Note que a configuração binária de ~a e a^b não são as mesmas. Ocorre que b ocupa dois bytes e a ocupa um só byte. Por isso os valores impressos são diferentes. Abaixo a representação em bits de a e b:

```
a = 00000001
b = 00000000 11111111
```

Operadores sobre sequências – listas, strings e tuplas

Sejam s e t de qualquer dos tipos acima. Em qualquer deles é possível:

Operação	Ação
s[j]	Acesso ao elemento j da sequência
s[inicio:fim]	Acesso a uma fatia destes elementos
s[inicio:fim:passo]	Acesso a uma fatia com deslocamento
s + t	Concatenção de s e t
k * s	O mesmo que s + s + s + ... (k vezes)
v in s	True se valor v é um dos elementos da sequência
v not in s	True se valor v não for um dos elementos da sequência

Os índices inicio, fim e passo podem ser positivos ou negativos indicando a direção em que devem ser considerados.

Podemos também comparar duas sequências com >, <, >=, <=, == e !=. São comparados elemento a elemento na ordem lexicográfica.

Veja os exemplos abaixo:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
b = "abcdefghij"
c = (1, -1, 2, -2, 3, -3)
# imprime com rotação
print(a[3:] + a[:3])
print(b[3:] + b[:3])
print(c[3:] + c[:3])
# imprime pulando um elemento
```

```
print(a[::2])
print(b[::2])
print(c[::2])
# imprime a sequência invertida
print(a[::-1])
print(b[::-1])
print(c[::-1])
```

Saída:

```
[4, 5, 6, 7, 8, 9, 1, 2, 3]
defghijabc
(-2, 3, -3, 1, -1, 2)
[1, 3, 5, 7, 9]
acegi
(1, 2, 3)
[9, 8, 7, 6, 5, 4, 3, 2, 1]
jihgfedcba
(-3, 3, -2, 2, -1, 1)
```

Remover um elemento de uma lista – remove, del e pop

```
# remove - remove o primeiro elemento igual da lista
a = [0, 2, 1, 2, 3, 2]
a.remove(2)
print(a)

# del - remove um elemento específico
a = [0, 2, 1, 2, 3, 2]
del a[2]
print(a)

# pop - removes o elemento de um índice específico
a = [0, 2, 1, 2, 3, 2]
a.pop(2)
print(a)

Saída:

[0, 1, 2, 3, 2]
[0, 2, 2, 3, 2]
[0, 2, 2, 3, 2]
```

Quando tentamos remover elementos que não existem, no remove o erro é “elemento não está na lista”, no del e no pop o erro é “índice inválido”.

Precedência de operadores

A tabela abaixo resumo a precedência dos operadores(fonte: Data Structures and Algorithms in Python – Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser) “[DAS - GTG]”

Operator Precedence		
	Type	**Symbols**
1	member access	expr.member
2	function/method calls container subscripts/slices	expr(...) expr[...]
3	exponentiation	**
4	unary operators	+expr, −expr, ~expr
5	multiplication, division	*, /, //, %
6	addition, subtraction	+, −
7	bitwise shifting	<<, >>
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	comparisons containment	is, is not, ==, !=, <, <=, >, >= in, not in
12	logical-not	not expr
13	logical-and	and
14	logical-or	or
15	conditional	val1 if cond else val2
16	assignments	=, +=, −=, *=, etc.

Funções

Definidas da seguinte forma:

```
def nome_da_função(<parâmetros>):
      <corpo da função>
```

Eventualmente a função não tem parâmetros.
A função pode devolver um valor do tipo simples (int, float, bool) ou sequência (listas, tuplas, sets).
O retorno do valor é feito pelo comando return, que tem o objetivo de retornar para onde houve a chamada da função e associar um valor à mesma.

```
return <valor>
```

Eventualmente a função termina sem que um comando return seja executado. Nesse caso, o valor de retorno é None.

Passagem de parâmetros

Os parâmetros presentes na definição da função são ditos parâmetros formais e os que estão presentes na chamada da função são ditos parâmetros atuais.

```
# definição da função
def nome_da_função(<parâmetros formais>):
      <corpo da função>
      ...
# chamada da função
...nome_da_função(<parâmetros atuais>)
```

Quando há uma chamada, é feita a associação entre os parâmetros atuais e os formais. Comportam-se como se fossem sinônimos. Entretanto, se algum parâmetro formal é alterado no corpo da função, outro objeto é criado e o correspondente parâmetro atual não é alterado. Neste aspecto, o mecanismo funciona como a passagem de parâmetros por valor da outras linguagens. Exemplo:

```
# definição da função
def func(x):
      ...
      x = -32 # o valor de z não é alterado

...
# chamada da função
z = 45
func(z)
```

Parâmetros Mutáveis

Quando o objeto passado como parâmetro é mutável (listas, sets, dicts), a função pode alterar o valor do parâmetro. Exemplo:

```
# zera uma lista
def zera(vt):
   for i in range(len(vt)):
         vt[i] = 0

vs = [1, 2, 3, 4]
zera(vs)
# vs voltará da função com valor [0, 0, 0, 0]
```

Como uma atribuição gera um novo objeto, a função abaixo não altera a lista.

```
# não esvazia a lista passada como parâmetro
def esvazia(vt):
   vt = []

vs = [1, 2, 3, 4]
esvazia(vs)
# vs continua com o valor [1, 2, 3, 4]
```

O exemplo seguinte, esvazia a lista. Usamos o comando del para eliminar cada elemento da lista.

```
# esvazia sim
def esvaziasim(vt):
    for i in range(len(vt)):
        del vt[0]

vs = [1, 2, 3, 4]
esvaziasim(vs)
# vs ficará vazia []
```

Outra forma para a função esvazia:

```
# outra esvazia
def esvaziasim(vt):
    del v[:]
```

Valor Default de parâmetro

Suponha uma função com a seguinte declaração:

```
def func(a, b = 15, c = -1):
    ...
```

Significa que os parâmetros b e c podem ser omitidos numa chamada. Quando assim forem, os seus valores default serão 15 e -1. A chamada func(3) é equivalente a chamada func(3, 15, -1). A chamada func(2, 5) é equivalente a chamada func(2, 5, -1). Uma consequência deste mecanismo é que se um parâmetro tem um valor default, os seguintes a ele também tem que ter. Podemos usar a comparação com None para verificar a ausência de um parâmetro.

Como ilustração, as funções intrínsecas int() e float() tem o valor zero se não há parâmetro:

```
k = int()          # k fica com valor 0
x = float()        # x fica com valor 0.0
```

Palavras-Chave como parâmetro

A definição:

```
def func(a = 0, b = 15, c = -1):
    ...
```

Possui todos os parâmetros com valores default. Se quisermos chamá-la apenas com o parâmetro c, a chamada será:

```
func(c = 33)
```

Com o nome do parâmetro evitamos colocar a sequência completa de parâmetros.

Expressões lambda

Quando uma função pode ser definida apenas como uma expressão, pode-se usar uma expressão lambda.

O comando:

```
somaquad = lambda x, y: x * x + y * y
```

é equivalente a:

```
def somaquad(x, y):
    return x * x + y * y
```

Podem ser usadas onde normalmente se usa uma função. Exemplos:

```
ar = [1, 2, 3, 4, 5, 6]
ssq = list(map(lambda x: 1 + x + x * x, ar))
print (ssq)

ssq = list(filter(lambda x: x % 2 == 0, ar))
print (ssq)
```

Saída:

```
[3, 7, 13, 21, 31, 43]
[2, 4, 6]
```

Funções intrínsecas mais comums do Python

A tabela abaixo mostra as funções intrínsecas mais comuns do Python (fonte: [DAS - GTG])

Common Built-In Functions	
Calling Syntax	**Description**
abs(x)	Return the absolute value of a number.
all(iterable)	Return True if bool(e) is True for each element e.
any(iterable)	Return True if bool(e) is True for at least one element e.
chr(integer)	Return a one-character string with the given Unicode code point.
divmod(x, y)	Return (x // y, x % y) as tuple, if x and y are integers.
hash(obj)	Return an integer hash value for the object (see Chapter 10).
id(obj)	Return the unique integer serving as an "identity" for the object.
input(prompt)	Return a string from standard input; the prompt is optional.
isinstance(obj, cls)	Determine if obj is an instance of the class (or a subclass).
iter(iterable)	Return a new iterator object for the parameter (see Section 1.8).
len(iterable)	Return the number of elements in the given iteration.
map(f, iter1, iter2, ...)	Return an iterator yielding the result of function calls f(e1, e2, ...) for respective elements e1 ∈ iter1, e2 ∈ iter2, ...
max(iterable)	Return the largest element of the given iteration.
max(a, b, c, ...)	Return the largest of the arguments.
min(iterable)	Return the smallest element of the given iteration.
min(a, b, c, ...)	Return the smallest of the arguments.
next(iterator)	Return the next element reported by the iterator (see Section 1.8).
open(filename, mode)	Open a file with the given name and access mode.
ord(char)	Return the Unicode code point of the given character.
pow(x, y)	Return the value x^y (as an integer if x and y are integers); equivalent to x ** y.
pow(x, y, z)	Return the value $(x^y \bmod z)$ as an integer.
print(obj1, obj2, ...)	Print the arguments, with separating spaces and trailing newline.
range(stop)	Construct an iteration of values $0, 1, \ldots,$ stop $- 1$.
range(start, stop)	Construct an iteration of values start, start $+ 1, \ldots,$ stop $- 1$.
range(start, stop, step)	Construct an iteration of values start, start + step, start + 2∗step, ...
reversed(sequence)	Return an iteration of the sequence in reverse.
round(x)	Return the nearest int value (a tie is broken toward the even value).
round(x, k)	Return the value rounded to the nearest 10^{-k} (return-type matches x).
sorted(iterable)	Return a list containing elements of the iterable in sorted order.
sum(iterable)	Return the sum of the elements in the iterable (must be numeric).
type(obj)	Return the class to which the instance obj belongs.

Tratamento de exceções

Durante a execução de um programa, erros inesperados podem acontecer (divisão por zero, uso de um índice inexistente de uma lista, tentativa de abrir um arquivo inexistente, uso de uma variável sem um valor associado, digitação de um caractere não numérico num pedido de entrada de números, etc.). Tais erros fazem com que o programa termine de uma maneira anormal e são chamados de exceções (exceptions).

O interpretador Python oferece uma forma de tratar esse tipo de erro se não queremos que o programa termine de forma anormal quando isso ocorre. O erro é interceptado pela execução de um trecho de programa específico.

As principais exceções que podem ser tratadas são mostradas na tabela abaixo (fonte: [DAS - GTG])

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Como interceptar uma exceção

Para interceptar a exceção usamos a construção try-except. No exemplo abaixo se queremos tratar especialmente o caso de uma possível divisão por zero.

```
try:
      ratio = x / y
except ZeroDivisionError:
      print("Tentativa de divisão por zero")
      ...
```

Outro caso é quando vamos abrir um arquivo. A abertura pode falhar por uma série de fatores (arquivo não existe, não temos a permissão adequada, arquivo de tipo diferente da solicitação de abertura, etc. Não é preciso especificar o tipo de exceção. Neste caso, qualquer delas será tratada. Quando especificado apenas esta será tratada:

```
# abrir o arquivo para leitura
try:
      arq = open(NomeArq, "r")
except:
      print("Arquivo não pode ser aberto")
```

Outro exemplo é proteger a entrada de dados contra a digitação de caracteres inválidos. Quando é solicitada uma entrada de int ou float por exemplo, a string digitada tem que ser compatível com o tipo. Veja abaixo a entrada de um dado float:

```
# loop até ler um float válido
while True:
    try:
        val = float(input("Entre com um float:"))
    except:
        # Entra neste ponto em caso de qualquer exceção.
        # Em particular quando algo errado é digitado.
        print("Entrada inválida")
    else:
        # Entra neste ponto se a digitação for correta
        print("Entrada válida")
        break
```

Quando, além do valor, é necessário evitar a digitação de dados não numéricos:

```
# Consistência de dados
# Entrar com um valor inteiro entre 0 e 99
while True:
    try:
        val = int(input("Entre com valor entre 0 e 99:"))
        if val < 0 or val > 99:
            print("Valor fora do intervalo 0 .. 99")
        else: break
    except (ValueError, EOFError):
        print("Valor digitado inconsistente")
```

Se necessário podemos imprimir mensagens específicas em cada uma das exceções:

```
# Consistência de dados
# Entrar com um valor inteiro entre 0 e 99
while True:
    try:
        val = int(input("Entre com valor entre 0 e 99:"))
        if val < 0 or val > 99:
            print("Valor fora do intervalo 0 .. 99")
        else: break
    except ValueError:
        print("Valor digitado inconsistente")
    except EOFError:
        print("Erro inesperada na entrada de dados")
```

Além dos blocos try, except e else, pode haver também um bloco finally que sempre é executado:

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("divisão por zero!")
    else:
        print("o resultado é:", result)
    finally:
        print("sempre executa o finally")

# testes
divide(3, 2)
divide(3, 0)
```

Os comandos acima imprimem:

```
o resultado é: 1.5
sempre executa o finally
divisão por zero!
sempre executa o finally
```

Outros usos para o try-except

Não é uma prática comum, mas pode-se usar também o try-except para contornar situações de erro em algoritmos. Exemplo:

```
# Procura x na lista a
def busca(a, x):
    try:
        i = 0
```

```
        # Se não encontrar vai dar índice inválido
        while a[i] != x: i += 1
        return i
    except:
        # Entra neste ponto se deu índice inválido
        return -1
```

Como sinalizar uma exceção

Através do comando raise. Suponha uma função que devolve a raiz quadrada de um valor que tem que ser maior ou igual a zero (a sqrt já faz isso):

```
def minha_sqrt(x):
    if x < 0:
        raise ValueError("O parâmetro não pode ser negativo")
    return sqrt(x)
```

Agora, em um programa ou função que use minha_sqrt, essa exceção pode ser interceptada.

O comando raise abandona a função. É um tipo de return que não retorna ao local onde foi feita a chamada,
Se não tratarmos a exceção, com a função acima, só vamos fazer com que a mensagem de exceção seja mais clara para o usuário. Os dois testes abaixo ilustram as duas maneiras de tratamento.

```
from math import sqrt
def minha_sqrt(x):
    if x < 0:
        raise ValueError('O parâmetro não pode ser negativo')
    return sqrt(x)

print("teste 1 - tratando a exceção")
try:
    print(minha_sqrt(-2))
except:
    print("Erro – parâmetro inválido")

print("teste 2 - sem tratar a exceção")
print(minha_sqrt(-1))
```

Saída:

```
teste 1 - tratando a exceção
Erro – parâmetro inválido
teste 2 - sem tratar a exceção
Traceback (most recent call last):
  File "C:/... /teste exceptions e mensagens.py", line 14, in <module>
    print(minha_sqrt(-1))
  File "C:/.../teste exceptions e mensagens.py", line 4, in minha_sqrt
    raise ValueError('O parâmetro não pode ser negativo')
ValueError: O parâmetro não pode ser negativo
```

A função sqrt real testa também se o tipo é numérico:

```
def sqrt(x):
      if not isinstance(x, (int, float)):
            raise TypeError("x deve ser numérico")
      elif x < 0:
            raise ValueError("x não pode ser negativo" )
      # calcule a raiz normalmente
      ...
```

Iterators e Generators – Iteradores e Geradores (tradução livre)

A forma geral do comando for é:

```
for <elemento> in <iterável> # tradução livre de iterable
```

Um objeto iterável (percorrível ! ! !) é uma lista, uma tupla, um set e um dic. Também são iteráveis strings (caractere a caractere). O que caracteriza a "iterabilidade" é que os elementos do objeto podem ser varridos sequencialmente do início ao fim, elemento a elemento.

Exemplos:

```
s = [45, 25, -12]
t = (1, -4, 82, 0)
c = {1, 4, -8, 120}
st = "exemplo"
d = {"x":21, "y":-78, "chato":1248}

# k é um iterador que varre os iteráveis

for k in s: print(k, " ", end = "")
print()
for k in t: print(k, " ", end = "")
print()
for k in c: print(k, " ", end = "")
print()
for k in st: print(k, " ", end = "")
print()
for k in d: print(k, " ", end = "")
print()
for k in d: print(d[k], " ", end = "")
print()
```

Saída:

```
45  25  -12
1  -4  82  0
-8  1  4  120
e  x  e  m  p  l  o
x  y  chato
21  -78  1248
```

A classe range() também é iterável. Importante notar que range(100) por exemplo não retorna uma lista de 100 elementos (0 a 99) e sim um objeto iterável diferente de lista. O mesmo ocorre com os demais objetos iteráveis. Se quisermos transformá-los em listas temos que usar a função list(). Veja exemplos abaixo:

```
s = [45, 25, -12]
t = (1, -4, 82, 0)
c = {1, 4, -8, 120}
st = "exemplo"
d = {"x":21, "y":-78, "chato":1248}

print(list(range(10)))
print(list(s))   # o mesmo que print(s)
```

```
print(list(t))
print(list(c))
print(list(st))
print(list(d))
print(list(d.values()))
print(list(d.keys()))
print(list(d.items()))
```

Será impresso:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[45, 25, -12]
[1, -4, 82, 0]
[-8, 1, 4, 120]
['e', 'x', 'e', 'm', 'p', 'l', 'o']
['x', 'y', 'chato']
[21, -78, 1248]
['x', 'y', 'chato']
[('x', 21), ('y', -78), ('chato', 1248)]
```

Generators – Geradores

Um objeto iterável pode ser criado também a partir de uma função do programa. O comando yield que não deve ser confundido com o comando return, produz cada uma dos elementos. O comando yield retorna da função, mas a próxima chamada continua no próximo comando a partir do yield.

```
# gerador dos números de fibonacci menores ou iguais a n
def fib(n):
    a = 0
    b = 1
    while a <= n:
        yield a
        a, b = b, a + b

# imprime os números de fibonacci menores ou iguais a 100
for x in fib(100): print(x)

# gera todos os anos bissextos do ano a1 até o ano a2
def bissexto(a1, a2):
    for k in range(a1, a2 + 1):
        if k % 4 == 0 and k % 400 != 0: yield k

# imprime os anos bissextos de 1980 a 2020
for x in bissexto(1980, 2020): print(x)
```

Comprehension syntax - Sintaxe resumida (tradução livre)

Python permite sintaxe resumida usando o comando for para facilitar a criação de sequências. Exemplos abaixo:

```
# vetor de n elementos com zeros
v = [0 for k in range(n)]
# matriz de n x m elementos zeros
mat = [m * [0] for k in range(n)]
```

```
# todos os fatores de N
fat = [k for k in range(2, N) if N % k == 0]

# set dos quadrados de 0 a t
ss = {j * j for j in range(t + 1)}
# dic dos quadrados de 0 a t
dd = {j:j * j for j in range(t + 1)}

# gerador dos quadrados de 0 a t-1
for k in (j * j for j in range(t)): print(k)
```

Escopo de nomes ou identificadores (Namespace)

Um comando de atribuição, sempre cria uma nova referência.

Quando é feita uma referência a uma variável dentro de uma função, a variável é procurada primeiramente nas variáveis que foram criadas nesta função. Se não for encontrada, a procura é feita nas variáveis no nível imediatamente superior. E assim por diante até o maior nível. O conceito do Python é que em cada ponto da execução existe um escopo para os nomes utilizados (Namespace).

```
a = 5
b = -1
def func():
    a = 10       # novo a
    print(a)
    print(b)     # b externo
    a = 20       # novo a
func()
print(a)         # a externo
```

Que imprime:

```
10
-1
5
```

Note agora que o exemplo abaixo imprime 5:

```
a = 5
def func():
    print(a)     # a externo

func()
```

Enquanto no exemplo abaixo teremos um erro (“UnboundLocalError: local variable 'a' referenced before assignment”) no comando print(a):

```
a = 5
def func():
    print(a)     # não considera externo porque há uma
    a = 3        # definição interna

func()
```

O motivo é que como a variável a é redefinida dentro da função, ela é considerada interna a essa função. Portanto não pode ser usada antes de receber um valor.

Identificadores globais (evite usar)

Sempre que necessário compartilhar valores entre funções, a melhor solução é passagem de parâmetros. Em raríssimos casos, é interessante o uso de identificadores globais como no exemplo abaixo. O atributo global, faz com que a referência a um identificador seja a externa:

```
a = 5
def func():
    global a    # muda a referência para o externo
    a = a + 1
    print(a)
    a = -32

func()
print(a)
```

Que imprime:

```
6
-32
```

Objetos especiais (first class objects)

Tipos (int, float, etc.), funções e classes, são considerados objetos especiais (first class) e podem ser passados como parâmetros ou atribuídos a outros nomes. Exemplo:

```
Mostre = print
...
Mostre("exemplo")     # mesmo que print("exemplo")
```

É a maneira para que funções sejam passadas também como parâmetros, como no exemplo abaixo:

```
from math import sin, cos, sqrt

def maxfunc(x, y, f = abs):
    " Devolve o máximo entre f(x) e f(y)
      o default de f é abs."
    a, b = f(x), f(y)
    if a > b: return x
    return y

# exemplos de chamadas
print(maxfunc(23, -45))
print(maxfunc(1, -1, sin))
print(maxfunc(10, -10, cos))
print(maxfunc(20, 32, sqrt))
```

Módulos e o comand import

Python tem entre 130 e 150 (depende da versão) funções e nomes intrínsecos (built-in). Além destes, existem milhares de funções e nomes, organizados em módulos, que podem ser

importados para serem usados por um programa. Como um exemplo, existem algumas funções matemáticas intrínsecas (abs, min, max, etc.) e muitas outras dentro do módulo math (sin, cos, sqrt, etc.) que podem ser importadas deste módulo.

A forma geral do comando import é:

```
from <módulo> import f1, f2, …
```

Dessa forma as funções são referenciadas apenas pelo seu nome f1, f2, etc.

Exemplo:

```
from math import sin, cos
```

Para importar todas:

```
from math import *
```

Outra forma é importar o módulo todo:

```
import math
```

Nessa forma a referência às funções deve ser precedida pelo nome do módulo: math.sin, math.cos.

Um novo módulo é simplesmente um arquivo com extensão .py. Exemplo: uma função fx que foi colocada no arquivo modulof.py pode ser importada com o seguinte comando:

```
from modulof import fx
```

O teste das funções do módulo

Quando o módulo todo é importado, são executados os comandos de sua parte principal se houverem. É comum quando se constrói um módulo, deixar junto dele um conjunto de comandos que testa as funções do módulo. Assim, toda vez que o módulo é atualizado, o teste está presente, inclusive para se certificar que as modificações introduzidas não alteraram o seu funcionamento anterior (testes de regressão).

Para evitar que esses testes ou outros possíveis comandos existentes no módulo sejam executados quando o mesmo é importado para ser usado por um outro programa, coloca-se o seguinte comando precedendo:

```
if __name__ == '__main__':
```

Módulos importantes (são parte da instalação básica)

Alguns módulos importantes existentes:

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

Módulos importantes (disponíveis, mas não são parte da instalação básica)

NumPy

É uma biblioteca de funções matemáticas usadas principalmente em cálculos com matrizes, ou arrays multidimensionais. A programação de algoritmos matemáticos, processamento de imagens e computação gráfica, algoritmos de machine learning é bastante facilitada com as funções desse módulo.

Numpy é um módulo bastante completo de funções. Alguns exemplos abaixo com arrays:

```
import numpy as np
# um vetor - array unidimensional
vetor = np.array([1, 2, 3, 4, 5, 6, 7])
print(vetor)
print()

# uma matriz de 3 linhas por 3 colunas
matriz = np.array([[1, 2, 3], [3, 2, 1], [0, 1, 0]])
print(matriz)
print()

# operações com matrizes
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

print(a + b)
print(a - b)
print(a * b) # multiplica termo a termo
print(a / b)
print()

# multiplicação de matrizes
print(a.dot(b))
```

Saída:

```
[1 2 3 4 5 6 7]
```

```
[[1 2 3]
 [3 2 1]
 [0 1 0]]

[[ 6  8]
 [10 12]]
[[-4 -4]
 [-4 -4]]
[[ 5 12]
 [21 32]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]

[[19 22]
 [43 50]]
```

A função np.array transforma uma lista num objeto array. A função dot, realiza a multiplicação algébrica de matrizes.

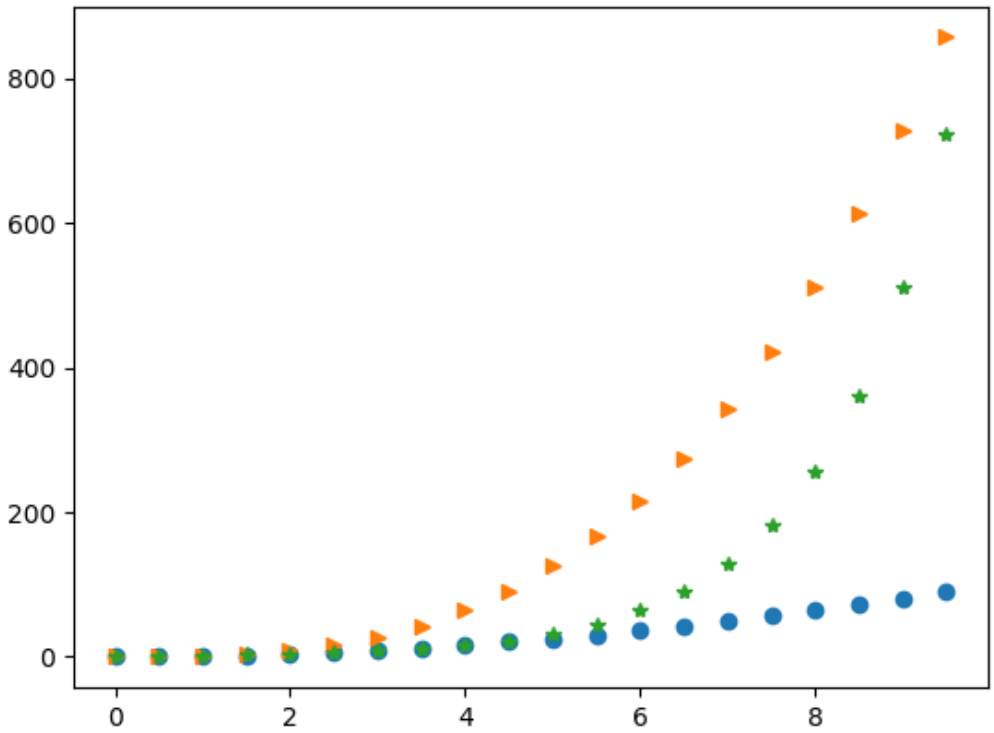
MatPlotLib

É uma biblioteca com funções para traçar gráficos. Vários tipos de gráficos são possíveis. Abaixo um exemplo simples mostrando as funções t^2, t^3 e 2^t no intervalo [0..10] com passo 0.5.

```
 import numpy as np
import matplotlib.pyplot as plt

# define as abcissas do gráfico. Intervalo [0..10] com passo 0.5
t = np.arange(0., 10., 0.5)

# círculos, setas e estrêlas
plt.plot(t, t**2, 'o', t, t**3, '>', t, 2**t, '*')
plt.show()
```



SciPy

É outra biblioteca com funções matemáticas, computação científica e engenharia. Inclui funções para tratamento de problemas de álgebra linear, interpolação de funções, otimização, estatística, integração numérica, transformadas de Fourier, processamento de sinais e processamento de imagens.

Pandas

É uma biblioteca com funções para análise e visualização de dados. Muito útil para construir planilhas e correlacionar os dados.