Python

É a linguagem da hora porque:

- ⋆ é moderna, usa e abusa de recursos de máquina.
- ⋆ é freeware
- ⋆ tem 170.000 pacotes, todos free também
- ⋆ usa bignumers, usa matemática complexa, ...
- ⋆ multiplataforma (mesmo)
- ⋆ fácil de aprender e usar
- ⋆ Unicode e UTF-8 por padrão
- ⋆ está no mainstream. Consulte www.tiobe.com

Pode ter certeza que usando Python, você estará em boa companhia

Instalação

Localize seu sistema operacional e capacidade de endereçamento do seus sistema (32 ou 64 bits) e daí vá em www.python.org escolha e instale a versão mais indicada. Sempre a 3, nunca a 2 que vai sair de linha.

Tipos básicos

`int`	123, -45, ...
`float`	12.3, -4.5, -1.5e-3...
`bool`	True ou False
`str`	'ola', "Curitiba", "let's go"

Os inteiros podem ser binários `0b010`, octais `0o654` ou hexadecimais `0xF5`. Dentro de strings, podem haver caracteres de escape como em `"aqui\nvamos`. Neste caso `\n` pula de linha.

Identificadores

nomes para variáveis, funções, módulos, classes,... `a..zA..Z seguido de a..zA..Z0..9_`

- diacríticos podem ser usados, mas não devem
- palavras reservadas proibidas
- minúsculas ≠ maiúsculas

Use nomes que façam sentido para você.

Assinalamento de variáveis

Usa `a=b`. A expressão `b` é avaliada e seu valor é associado ao nome `a`.

`x=1+6.1+cos(z)`	
`a=b=c=0`	as 3 variáveis recebem zero.
`a,b,c=3,5,7.8`	a recebe 3, b 5 e c 7.8.
`a,b=b,a`	a e b trocam o valor
`a,*b=lista`	Separa lista em cabeça e cauda
`a+=1`	Incrementa a em 1
`del a`	Remove a variável a

Matemática

Operações: `+`, `-`, `*`, `**` (potência), `/` (divisão real), `//` (divisão inteira), `%` (resto). `10/3` é `3.33333333` enquanto `10//3` é `3`. Pode ser que se exija a importação do módulo `math`.

adição	`a+b`	2+3 → 5
subtração	`a-b`	2-3 → -1
multiplicação	`a*b`	3*5→ 15
divisão real	`a/b`	10/3→ 3.33333
div. inteira	`a//b`	10//3→3
potência	`a**b`	2**3→8
resto	`a%b`	10%3→1
módulo	`abs(a)`	abs(-3)→3
arredond.	`round(a)`	round(3.67,1)→3.7
seno	`sin(a)`	sin(pi/4)→0.707...
cosseno	`cos(a)`	cos(2*pi/3)→ 0.49
tangente	`tan(a)`	tan(1)→1.55...
$\sqrt[2]{a}$	`sqrt(a)`	sqrt(2)→1.414...
$log_e(a)$	`log(a)`	log(e**2)→2
teto	`ceil(a)`	ceil(3.1)→4
chão	`floor(a)`	floor(3.6)→3

Entrada e Saída

```
aa=int(input('Forneça o valor: '))
```

O programa pára, imprime a mensagem Forneça o valor: aguarda o usuário digitar algo e este valor, devidamente transformado em inteiro é armazenado na variável aa. A resposta com comando input SEMPRE volta no formato string, pelo que a conversão exata deve ser claramente estabelecida.

```
a=1;b=5
print('x=',a,' c=',b)
  # vai imprimir x= 1  c= 5
```

Tipos de contenedores

`lista` sequência ordenada, acesso por índice. Reconhecida pelo `[ , ]`. Exemplo `[1,4,9]`, `['alfa',3,0.99]`, ...

`dicionário` tabela de chave:conteúdo, acesso por chave. Reconhecida por `{chave:conteudo, ,}`. Exemplo `{1:"paraná", 2:"piauí", ...}`.

`Conjunto` dados não ordenados, sem duplicações. Acessados por operações de conjuntos. Reconhecido por `{}` ou `set()`. Exemplo `{1,3,77,90}`

`tuplas` listas fixas, que não podem sofrer alteração. Reconhecidas por `( , )`. Exemplo `(1,2,3)`.

Conversões

`int("123")`	`o valor 123`
`int("2F",16)`	38
`int(12.3)`	12
`float(1)`	1.0
`float(-3.23e4)`	-32300.0
`round(1.7)`	2
`chr(64)`	@
`ord('@')`	64
`bytes([52,65,67])`	`b'4AC'`
`list("IJK")`	`['I', 'J', 'K']`

```
dict([(1,"um"),(2,'dois')])
→       {1:'um',2:'dois'}
set('um','do')→{'um','dois'}
'*'.join(['um','do'])→'um*do'
'um dois 3'.split()
→       ['1', 'dois', '3', '4']
[int(x) for x in('1','33','267')]
→       [1, 33, 267]
```

Lógica Booleana

Comparadores em Python: `==`, `!=`, `>=`, `<=`, `>`, `<`. Conectores: `and`, `or` e `not`. Constantes booleanas: `True` e `False`.

Fatiamentos

Para listas, tuplas, strings, bytes,... mas não para dicionários e conjuntos.

a=	[	10	,	20	,	30	,	40	,	50	]
índs neg		-5		-4		-3		-2		-1	
índs pos		0		1		2		3		4	
fatia pos	0		1		2		3		4		5
fatia neg	-5		-4		-3		-2		-1		

```
len(a)→5
a[0]  →10
a[-1] →50
a[1]  →20
```

Agora, sempre no formato `[inicio:até fim:increment`

```
a[:-1] →[10, 20, 30, 40]
a[1:-1]→[20, 30, 40]
a[::2] →[10, 30, 50]
a[::-1]→[50, 40, 30, 20, 10]
a[::-2]→[50, 30, 10]
a[:]   →[10, 20, 30, 40, 50]
          (cópia rasa da seq)
a[1:3] →[20, 30]
a[-3:-1]→[30, 40]
a[:3]  →[10, 20, 30]
a[3:]  →[40, 50]
```

Importação de módulos

No comando `import aaa`, o Python vai buscar o módulo `aaa.py`. Note que o S.O. precisa achar o arquivo (alterar PATH, por exemplo).

Em `import aaa`, o elemento de nome `bbb` do módulo vai ser acessado escrevendo `aaa.bbb`.

Em `import aaa as xxx`, o elemento de nome `bbb` do módulo vai ser acessado escrevendo `xxx.bbb`.

Em `from aaa import *`, o elemento de nome `bbb` do módulo vai ser acessado escrevendo `bbb`.

Condicional

```
if condição:
   executado se cond==True
```

O bloco indentado só é executado se a condição for verdadeira.

```
if condição:
   bloco-1
else:
   bloco-2
bloco-3
```

```
Exemplo:
if x<6:
    print('a')
else:
    print('b')
print('c')
```

O bloco1 é executado se a condição é True. Se a condição é False, executa-se o bloco2. Em qualquer um dos casos, bloco3 é executado incondicionalmente. No exemplo, se $x = 5$ imprimem-se as letras a e c.

Há ainda o comando elif, veja

```
if a<=100:
   bloco-1
elif a<=500:
   bloco-2
elif a<=1000:
   bloco-3
else:
   bloco-4
continuação
```

Se $a < 100$, executa-se o bloco1 e depois a continuação. Se $100 < a \leq 500$ executa-se o bloco2 e depois a continuação. Se $500 < a \leq 1000$ executa-se o bloco3 e depois a continuação. Se $a > 1000$ executa-se o bloco 4 e depois a continuação.

Repetição

```
while condição:
   bloco-1
bloco-2
```

O bloco1 será executado repetidas vezes, enquanto a condição for verdadeira. Ao final o bloco2 será normalmente executado.

Saída de repetição

Existem 2 saídas mandatórias para laços while e for. São: `break` que determina a saída imediata do laço. Há também `continue` que força o reexame da condição de saída.

Interadores

Os interadores estão em python como os indexadores estão em C, mas são muito mais abrangentes. Além de ser elementos em um vetor, podem ser elementos em uma lista, tupla, dicionário, conjunto ou registro em um arquivo, entre outros. Exemplos:

```
a={1:300,5:400,7:500}
for i in a:       # impressos:
  print(i)        # 1, 5 e 7
a=(1,20,300)
for i in a:       # impressos:
  print(i)        # 1, 20 e 300
a={1,6,8}
for i in a:       # impressos:
  print(i)        # 1, 6 e 8
a=[1, 6, 8, 9, 11, 20]
print([x for x in a if x%2==0])
    # impressos [6,8,20]
[x**2 for x in range(5)]
 #imprime [0, 1, 4, 9, 16]
```

Operações em contenedores

As operações matémáticas (sum, min,...) exigem dados numéricos.

tamanho	`len(c)`	len([1,2,3])→ 3
mínimo	`min(c)`	min([3,2,7])→ 2
máximo	`max(c)`	max([3,2,7])→ 7
soma	`sum(c)`	sum([1,2,3])→ 6
sort	`sort(c)`	sort([3,2,7])→[2,3,7]
pertença	`x in c`	3 in [3,2,7]→ True
todos	`all(c)`	verdadeiro se todos os c são verdadeiros
algum	`any(c)`	verdadeiro se algum c é verdadeiro
reverso	`reversed(c)`	c em ordem reversa
posição	`c.index(v)`	índice de v em c
contador	`c.count(v)`	quantas ocorrências de v em c

Operações em listas

inserção	`L.append(v)`	insere v ao final de L
inserçao múltipla	`L.extend(p)`	insere a lista p no final de L
inserção	`L.insert(i,v)`	insere v após o índice i em L
remoção	`L.remove(v)`	remove a 1^o ocorrência de v em L
pop	`L.pop([i])`→ v	remove e retorna $L[i]$
ordena	`L.sort()`	ordena *in place*
reverte	`L.reverse()`	reverte *in place*

Operações em dicionários

chave	`d[chave]=v`	atualiza a chave de d com v
chave	`d[chave]` →v	consulta a chave de d
limpa	`D.clear()`	limpa o dicionário D
exclui	`del D[chave]`	elimina a entrada *chave*
atualiza	`D.update(d2)`	atualiza a entrada $d2$
chaves	`D.keys()`	chaves interadas
valores	`D.values()`	valores interadas
pares	`D.items()`	pares *chave*, *valor* interadas

Operações em conjuntos

união	`|`	conjunto união
intersecção	`&`	conjunto intersecção
diferença	`-`	a={1,4,5,6,8,9}, b={2,3,4,5}, $a-b \rightarrow \{8,1,6,9\}$
dif. simétrica	`^`	`a^b`→ $\{1,2,3,6,8,9\}$
adição	`S.add(v)`	Adiciona elemento v no dicionário S
exclusão	`S.remove(v)`	remove v no dicionário S
limpeza	`S.clear()`	zera o dicionário S

Arquivos

abertura	`f=open(a,t)`	abre o arquivo a e cria a variável f. t pode ser w, r ou a
leitura	`a=f.read(n)`	Lê n caracteres de f e joga em a
escrita	`f.write(q)`	grava q no arquivo f
le linha	`f.readline()`	lê a próxima linha em f
le linhas	`f.readlines(n)`	lê n próximas linhas em f
fechamento	`f.close()`	Fecha o arquivo f
aplica	`f.flush()`	apliaca mudanças em f

```
def arq():
  f=open('c:/apl2/vivoblo1',"r")
  for linha in f:
    print(linha)
arq()
```

Sequências inteiras

```
range([inicio,]fim[,etapa])
```

`início` se omitido, vale 0. `etapa` se omitido, vale 1. `fim` encerra antes deste valor.

`range(5)`	→[0,1,2,3,4]
`range(3,8)`	→[3,4,5,6,7]
`range(2,12,3)`	→[2,5,8,11]
`range(20,5,-5)`	→[20,15,10]

Definição de funções

```
def numefun([parâmetros]):
   """documentação"""
   ...computações...
   return resultado
```

Os parâmetros e todas as variáveis definidas na função só existem enquanto a função é executada. São portanto locais. O comando `return` encerra a execução da função. Comandos após ele nunca serão executados. Após a definição, a função pode ser chamada, como em

```
aa = nomefun(x,y,z)
```

Seu resultado será armazenado em aa e as variáveis x, y, z serão usadas no lugar dos parâmetros usados lá na definição. Os parênteses indicam que é uma função.

Operações em strings

S.split([e])	Cria lista, separando pelo e [espaços]
[j].join(L)	junta elementos de L, conectando-os com j
S.strip([c])	Remove c ou brancos no início e fim de S
S.count('c')	Conta quantos c há em S
S.partition(e)	Separa em 3 pedaços: antes, e e depois
S.find(u,i,f)	Retorna o primeiro u começando em i, terminando em f ou -1
S.is...()	... pode ser alpha... Retorna True se sim
S.upper()	Retorna maiúsculas
S.lower()	Retorna minúsculas
S.title()	Só as iniciais maiúsculas
S.swapcase()	troca a caixa
S.center(t[,f])	centraliza em t preenchendo com f
S.startswith(p,i,f)	True se S começa com p
S.endswith(u,i,f)	True se S termina com u

Formatação

`"padrão".format(x,y,z...)`→string padrão é formado por diversas {}, cada uma podendo ter: seleção:formato!conversão. seleção é a sequência de variáveis. Se omitida, vale a mesma ordem em que aparecem. formato pode ter: caráter de preenchimento alinhamento sinal tamanho-mínimo tamanho-máximo tipo. tipo pode ser `bcdoxXeEfFgGs%`. Conversão pode ser `sr`. Eis alguns exemplos

```
from datetime import *
print('{} {}'.format(1,2))   #  1b2
print('{} {}'.format(1,2.1)) #  1b2.1
print('{1}  {0}'.format(1,2)) #  2bbb1
print('{:f}'.format(3.14159265))
     # 3.141593
print('{:d}'.format(2))  # 2
print('{:5d}'.format(2)) # bbbb2
print('{:05d}'.format(2)) # 00002
print('{:06.2f}'.format(3.1415))
     # 003.14
print('{:%Y-%m-%d %H:%M}'.format
 (datetime(2019, 2, 3, 4, 5)))
 #2019-02-03 04:05
print('{1:3d}&{0:3.1f}'
 .format(1,2)) #bb2&1.0
```

Objetos/Classes

Objetos são a soma de estrutura de dados + métodos. São perfeitos para descrever o universo e encapsular a complexidade.

```
class Carro:
  def __init__(self, marca, ano):
     #construtor
    self.marca=marca
    self.ano=ano
  def metodo(self):
    print("a marca e ",self.marca)
class Colecao(Carro):
     # colecao herda Carro
  def __init__(self,m,a,c):
    Carro.__init__(self,m,a)
    self.cor=c
c1=Carro("fusca",1970)
c1.metodo()
c2=Colecao("chevete",1973,"branco")
```