Variáveis e Tipos Embutidos

Variáveis no interpretador Python são criadas através da atribuição e destruídas pelo coletor de lixo (***garbage colector***), quando não existem mais referências a elas.

Os tipos e rotinas mais comuns estão implementados na forma de ***builtins***, ou seja, eles estão sempre disponíveis em tempo de execução, sem a necessidade de importar nenhuma biblioteca.

Tipos Embutidos (Built-ins)

Um **valor**, como um **número** ou **texto**, é algo comum em um programa. Por exemplo, 'Hello, World', 1, 2 todo são valores.

Caso você não tenha certeza qual é o tipo de um valor, pode usar a função `type()` para checar:

```
type('Hello World')
<class 'str'>
```

```
type(2)
<classe 'int'>

type('3.2')
<class 'str'>

type(2+3j)
<classe 'complex'>
```

Operadores e Operações matemáticas

Operador	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão (com resultado fracionário)
//	Divisão (sem resultado fracionário)
%	Módulo ou resto
**	Exponenciação

Os parênteses são utilizados em Python da mesma forma que em expressões matemáticas. A **ordem de precedência** fica:

1. Exponenciação;
2. Multiplicação, Divisão e Módulo;
3. Adição e Subtração.

Conceitos de variáveis e atribuição

As **variáveis** são utilizadas para armazenar valores e para dar nome a uma área de memória do computador onde armazenamos dados. Para armazenar algo na

memória, usaremos o símbolo de igualdade (=), operação de **atribuição**. Vamos escrever:

```
# Primeiro programa com variáveis
a = 2
b = 3
print(a+b)
```

O **símbolo** `#` é usado para indicar que estamos comentando ou fazendo um comentário. **Comentários** são ignorados pelo interpretador Python. Vamos escrever outro programa:

```
# Cálculo de aumento de salário
salario = 1500
aumento = 5
print(salario + (salario*aumento/100))
```

Números

Além dos números inteiros convencionais, existem também os inteiros longos, que tem dimensão arbitrária e são limitados pela memória disponível. As conversões entre inteiro e longo são realizadas de forma automática. A função
builtin `int()` pode ser usada para converter outros tipos para inteiro, incluindo mudanças de base.

```
#convertendo de real para inteiro
print('int(3.14)=', int(3.14))

#convertendo de inteiro para real
print('float(5)=', float(5))
```

```
#cáculo entre inteiro e real resulta em real
print('5.0/2+3 =',5.0/2+3)

#inteiros em outra base
print("int('20',8) =", int('20',8)) #base 8
print("int('20',16) =", int('20',16)) #base 16

#operações com números complexos
c=3+4j
print('c=', c)
print('Parte real:', c.real)
print('Parte imaginária:', c.imag)
print('Conjugado:', c.conjugate())
```

Variáveis String

Variáveis do tipo **string** armazenam cadeias de **caracteres** como nomes e textos em geral.

Operador "%" é usado para fazer interpolação de *strings*. A interpolação é mais eficiente no uso de memória do que a concatenação convencional.

Símbolos usados na interpolação:

Especificador	Representa
%s	um string
%o, %d, %x	um número inteiro em octal, decimal ou hexadecimal
%e	um real exponencial
%f	um número real de precisão simples
%%	um único sinal de porcentagem

Um string em Python tem um tamanho associado, assim como um conteúdo que pode ser acessado caractere a caractere. O tamanho de uma string pode

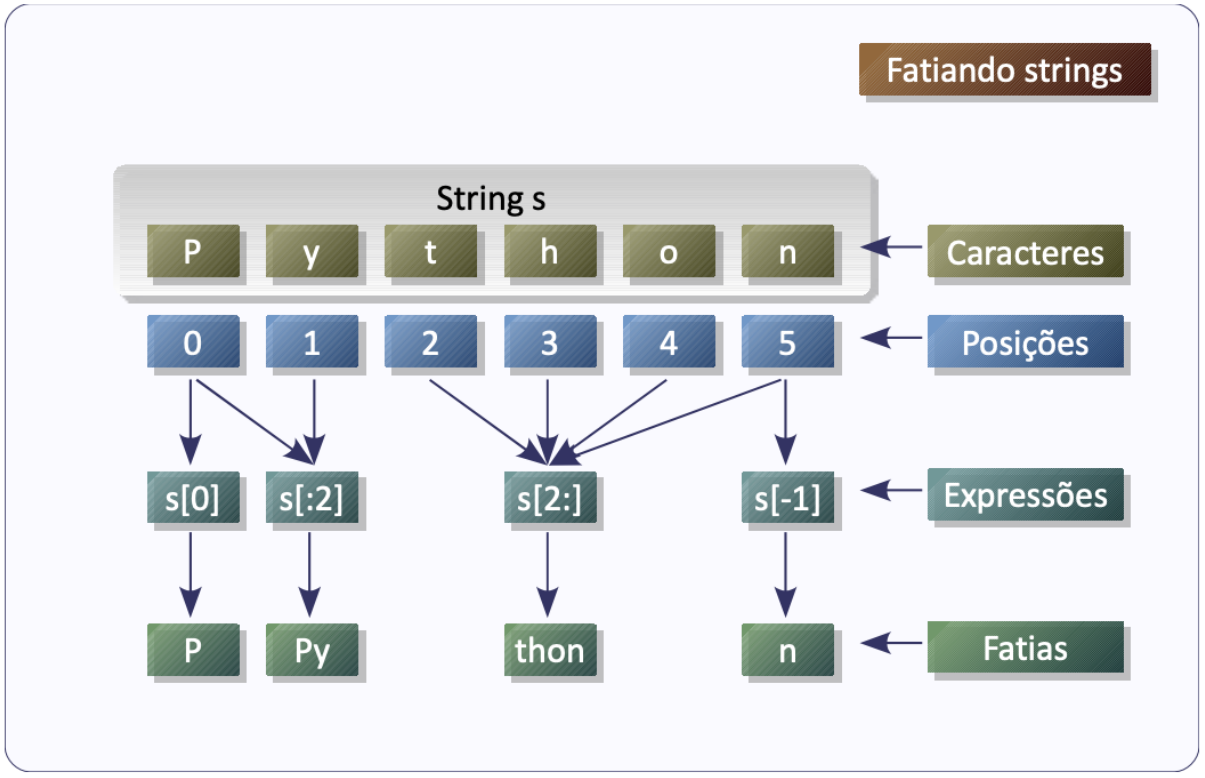
ser obtido utilizando-se a função `len()`.

```
len("O rato roeu a roupa")
19
```

```
s = 'Ferrari'

# Concatenação
print('O carro '+s+' está na estrada') # Interpolação

# String tratada como sequência
print('O tamanho de %s => %d' %(s, len(s)))

for ch in s: print(ch) # Strings são objetos

if s.startswith('F'): print(s.upper())
```

Para acessar os caracteres de uma string, devemos informar o índice ou posição do caractere entre colchetes `[]`. Como o primeiro caractere de uma string é o de **índice 0**, podemos acessar valores de 0 até o tamanho da string menos 1.

```
a = "ABCDEF"
a[5]
F
a[2:4]
CD
a[:2]
AB
a[1:]
BCDEF
```



Os índices no Python:

- Começam em zero.
- Contam a partir do fim se forem negativos.
- Podem ser definidos como trechos, na forma [inicio:fim + 1:intervalo]. Se não for definido o inicio, será considerado como zero. Se não for definido o fim + 1, será considerado o tamanho do objeto. O intervalo (entre os caracteres), se não for definido, será 1.

É possível inverter
strings usando um intervalo negativo:

```
print('Python'[::-1])
#Mostra: nohtyP
```

O operador + e o operador * também funcionam com strings:

```
texto1 = 'oi'
texto2 = 'Python'
print(texto1+texto2)
print(texto2*3)
```

O resultado será:

oiPython

Python Python Python

O método `upper()` retorna o texto em letras maiúsculas e o método `capitalize()` retorna o texto com a primeira letra em maiúscula.

```
texto1 = 'python'
texto1.upper()
texto1.capitalize()
```

Sairá:

PYTHON

Python

Entrada de Usuário

O Python possui uma função que captura a entrada de valores: a função `input()`.

```
nome=input("Digite seu nome: ")
print(nome)
```

Podemos melhoras o código:

```
nome = input("Digite um nome: ")
idade = input("Digite sua idade: ")
print("Seu nome é {} e tem {} anos".format(nome, idade))
```

```
nome = input("Digite um nome: ")
idade = input("Digite sua idade: ")
print(f"Seu nome é {nome} e tem {idade} anos")
```

A função `format()` vai substituir o `{}` pela variável `nome` e `idade`.

```
musicos = [('Page', 'guitarrista', 'Led Zeppelin'),
('Fripp', 'guitarrista', 'King Crimson')]

# Parâmetros identificados pela ordem
msg = '{0} é {1} da {2}'

for nome, funcao, banda in musicos:
    print(msg.format(nome, funcao, banda)) # Parâmetros ident


msg = '{saudacao}, são {hora:02d}:{minuto:02d}'
print(msg.format(saudacao='Bom dia', hora=7, minuto=30))

# Função builtin format()
print('Pi =', format(3.14159, '.3e'))
```

Conversão da entrada de dados

A função `int()` é utilizada para converter o valor retornado em número inteiro, a função `float()` converte em número decimal.

```
anos = int(input("Anos de serviço: "))
valor_por_ano = float(input("Valor por ano:"))
```

F-Strings

Já utilizamos **F-Strings** para compor strings, mas vamos revisitar o assunto. F-Strings foram introduzidas na versão 3.6 do Python. Com essa sintaxe, é possível substituir o valor de uma variável ou expressão dentro de uma string. Por exemplo:

```
a="mundo"
printf(f"Alô {a}")
```

É equivalente a `“Alô %s" %a` ou `“Alô {}.format(a)”` .

Você pode também formatar **f-strings** especificando o número de caracteres após o nome da variável e dos dois pontos.

```
preco =5.20
f"Preço: {preco:5.2f}"
f"Preço: {preco:10.2f}"
f"Preço: {preco:.2f}"
```

Você também pode usar `>`, `<` e `^` para alinhar os valores à esquerda, à direita ou ao centro:

```
f"Preco: R${preco:>10.2f}!"
f"Preco: R${preco:<10.2f}!"
f"Preco: R${preco:^10.2f}!"
```

E também especificar qual caractere deve ser utilizado para preencher os espaços em branco:

```
f"Preco: R${preco:.^10.2f}!"
f"Preco: R${preco:x^10.2f}!"
f"Preco: R${preco:_^10.2f}!"
```

Essa nova forma de escrever é tão poderosa que você pode até chamar funções dentro da **f-string**:

```
x = 5.1
f"Inteiro: {int(x)}"
f"Inteiro: {int(x)*10: 5.2f}"
```

Constantes

O Python possui poucas constantes embutidas. As mais utilizadas são **True**, **False** e **None**. **True** e **False** são valores **booleanos**. Python também possui a função `bool()`, que retorna **True** quando o argumento passado é verdadeiro e retorna **False**, caso contrário.

```
2>1
True
```

```
bool(3>5)
False
```

O **None** é um valor **NoneType**, e é usado para representar a abstenção de um valor. Em outras linguagens de programação, é comum utilizar a palavra **Null** para representar a abstenção.

Operadores de Comparação

Operação	Descrição	Operação	Descrição
a == b	a igual a b	a is b	True se a e b são idênticos
a != b	a diferente de b	a is not b	True se a e b não são idênticos
a<b	a menor do que b	a in b	True se a é membro de b
a>b	a maior do que b	a not in b	True se a não é membro de b
a<=b	a menor ou igual a b	a and b	True se os dois forem verdadeiros
a>=b	a maior ou igual a b	a or b	Falso apenas quando os dois são falsos

O operador `==` checa se o conteúdo das variáveis são **iguais**. Já o operador `is` checa se a e b são o **mesmo objeto**.

Zen Python

Escreva no interpretador o seguinte comando:

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```