

PYTHON PARA PLN

Introdução ao Python

Roney Lira de Sales Santos roneysantos@usp.br

Rogério Figueredo de Sousa rogerfig@usp.br

Prof. Thiago A. S. Pardo

INTRODUÇÃO À LINGUAGEM

- Linguagem de **alto nível**
 - Ao ler o comando, já se presume o que ele significa!
- Vários tipos de programação:
 - Modular: **divisão** do algoritmo em blocos
 - Orientada a objetos: **classes** e **objetos** referenciados
 - Funcional: aplicação de **funções matemáticas**
- Tipagem **forte** e dinâmica
- Grande coleção de bibliotecas
- Código aberto

INTRODUÇÃO À LINGUAGEM

- Instalação e uso: site oficial do Python
 - <https://www.python.org/downloads/>



- Os códigos podem ser executados:
 - Em programas como **PyCharm**, **VS Code** ou **Sublime**
 - por meio de linha de comando (prompt Windows, terminal Linux, IDLE Python...)
 - `python programa.py`
 - `./programa.py`

TIPOS DE DADOS

```
>>> x
10
>>> y = "roney lira"
>>> y
'roney lira'
>>> k = 1.8 - 1.5
>>> k
0.30000000000000004
>>> type(x)
<class 'int'>
>>> type(y)
<class 'str'>
>>> type(k)
<class 'float'>
>>> x > 6
True
>>> x > 15
False
>>> |
```

- Tipos de dados básicos
 - int, string, float, bool...
- O **tipo** de uma variável **muda** conforme o valor que lhe é atribuído.
 - Princípio da dinâmica
- **type (var)**

INTRODUÇÃO À LINGUAGEM

- Não é preciso terminar comandos com ;
- Não é preciso **declarar o tipo de dados** das variáveis

```
>>> x = 4+6
>>> x
10
>>> y = "roney lira"
>>> y
'roney lira'
>>> k = 1.8 - 1.5
>>> k
0.30000000000000004
>>> type(x)
<class 'int'>
>>> type(y)
<class 'str'>
>>> type(k)
<class 'float'>
>>> |
```

TIPAGEM FORTE

```
>>> v = "1"
>>> q = 1
>>> v + q
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    v + q
TypeError: can only concatenate str (not "int") to str
>>> |
```

- Ou seja, Python é uma linguagem que tem a característica de que não permite um mesmo dado ser tratado como se fosse de outro tipo.
- Mais sobre isso [aqui](#).

INDENTAÇÃO

- Bom, Python não usa nenhum limitante para estrutura de bloco, comum em outras linguagens
 - {, }, ;, **begin**, **end**, e por aí vai.
- A estrutura do bloco é definida pela indentação, ou seja, o alinhamento dos comandos é que define a estrutura!

```
if num % 2 == 0:  
    par = par + 1  
    print("PAR")  
else:  
    impar = impar + 1  
    print("IMPAR")
```

```
14 if num % 2 == 0:  
15 print("PAR")
```

```
File "C:\Users\roney\OneDrive\Documentos\10.  
PLN\aula8.py", line 15  
    print("PAR")  
    ^
```

```
IndentationError: expected an indented block  
[Finished in 0.4s]
```

PARA NOSSAS PRÁTICAS, USAREMOS...

- Listas e Tuplas
- Dicionários
- Arquivos
- Strings
 - Textos, cara do nosso PLN né! =)
- Claro, existem beeeem mais coisas relacionadas ao Python!
 - Estruturas de controle, funções, classes, programação funcional, uso de *frameworks* diversos...
- E também *frameworks* relacionados ao PLN, como **NLTK** e **spaCy**
 - **Em breve** nas telinhas da disciplina!

LISTAS E TUPLAS

- Estruturas de dados **nativas** da linguagem
 - **list** e **tuple**
- Informações dentro das listas e tuplas podem ser de tipos diferentes
- Acesso sequencial: fatias (*slicing*) ou diretamente
- Métodos prontos para adicionar, remover, ordenar, procurar, contar, entre vários outros
- **Listas**: **mutáveis** e delimitadas por **colchetes**
- **Tuplas**: **imutáveis** e delimitadas por **parênteses**

LISTAS - MÉTODOS

```
l = list(range(5))
print(l)
l.append(5)
print(l)
l.insert(0,6)
print(l)
l.reverse()
print(l)
l.sort()
print(l)
```

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[6, 0, 1, 2, 3, 4, 5]
[5, 4, 3, 2, 1, 0, 6]
[0, 1, 2, 3, 4, 5, 6]
```

- **extend (L)** : inclusão de uma lista **l2** (**append**)
- **remove (x)** : remove a primeira ocorrência de **x**
- **index (x)** : índice da primeira ocorrência de **x**
- **count (x)** : número de ocorrências de **x** na lista

LISTAS - SLICE

- Funciona como uma **sublista** da lista

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:7]
[2, 3, 4, 5, 6]
```

- A notação **[2:7]** significa qual o **intervalo da lista original** você pretende retornar
 - Lembrando que os índices são contados **a partir do zero**. Assim, no nosso exemplo, queremos do 3º elemento até o 7º.

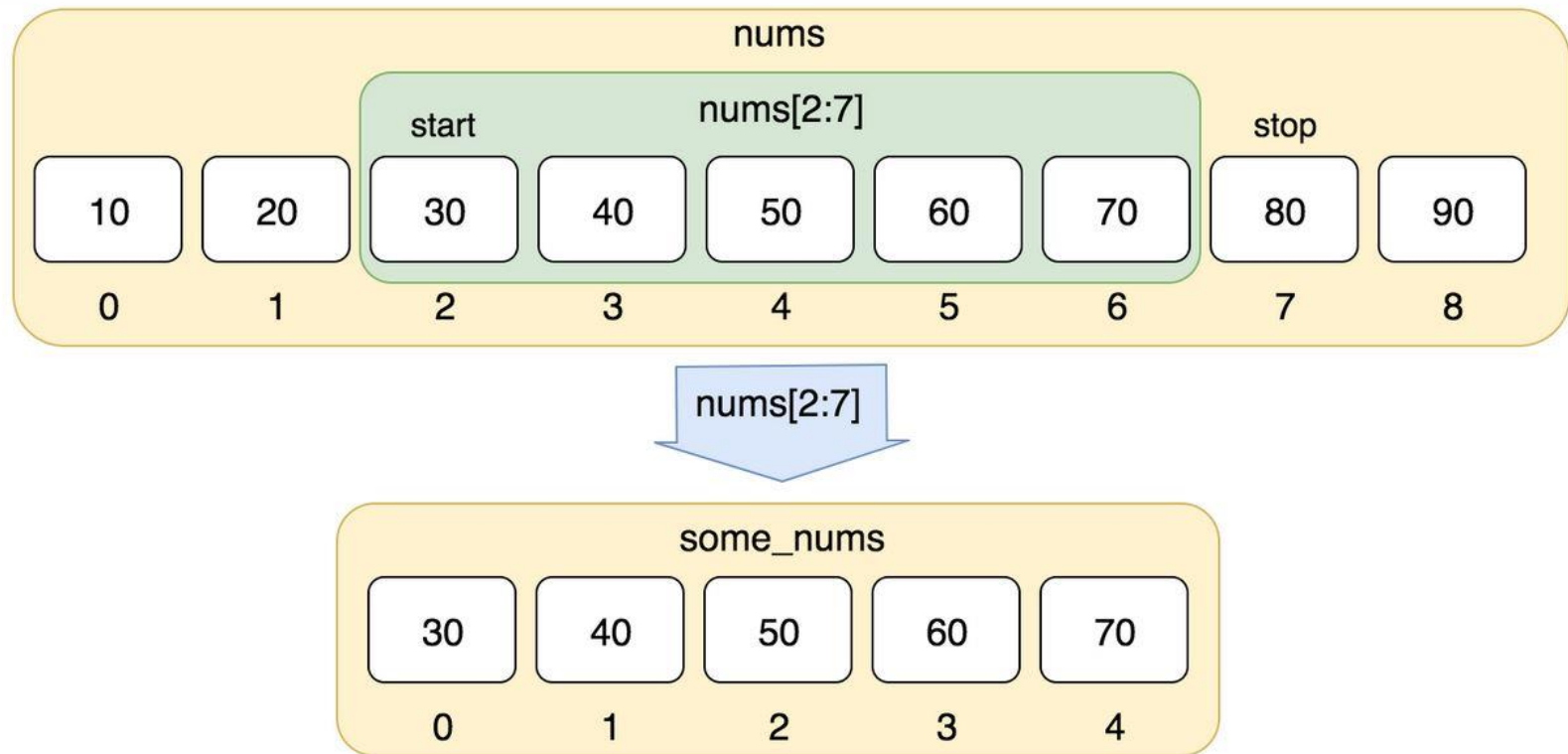
LISTAS - SLICE

- Também é possível **pular elementos**, incluindo a quantidade em um outro índice.

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:7]
[2, 3, 4, 5, 6]
>>> l[2:7:2]
[2, 4, 6]
```

- A notação **[2:7:2]** significa, então, que queremos uma sublista do 3º ao 7º elemento, **pulando de 2 em 2 elementos**.

LISTAS - SLICE



LISTAS - SLICE

- Conseguimos também retornar os **n** primeiros elementos e os **n** últimos elementos por meio do *slicing*.

- Os **n** primeiros elementos: **[:n]**

- Ou seja, deixa-se vazio o primeiro index da lista

```
>>> l[:5]
[0, 1, 2, 3, 4]
```

- Os **n** últimos elementos: **[-n:]**

- Deixa-se vazio o segundo index da lista

```
>>> l[-5:]
[5, 6, 7, 8, 9]
```

LISTAS - SLICE

- Além disso, várias outras combinações podem ser feitas...
- Retornar todos os elementos **menos os n últimos**:
 - `l[:-n]`
- Pular de **n em n** elementos na lista:
 - `l[::n]`
- O **reverse ()** pode ser representado por `l[::-1]`
- E vááárias outras possibilidades!
 - Dá uma olhada [aqui](#), ó!

TUPLAS

- Tuplas seguem o mesmo conceito das listas, porém com uma diferença importante: são **imutáveis**.
- Existe uma outra diferença: tuplas são para elementos **heterogêneos**, ou seja, tipos diferentes dentro da tupla.
 - Porém, como o Python é uma linguagem dinâmica, essa característica também aparece nas listas.
 - Cabe **ao programador decidir** tal característica.

TUPLAS

- Por exemplo, se você tentar **modificar um elemento** da tupla, você não consegue. Exemplo:

```
>>> t = (1, 'roney', 2, 'lira')
```

```
>>> t
```

```
(1, 'roney', 2, 'lira')
```

```
>>> t[0] = 5
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#5>", line 1, in <module>
```

```
    t[0] = 5
```

```
TypeError: 'tuple' object does not support item assignment
```

```
>>> |
```

TUPLAS

- Então você pode perguntar: **Mas então, qual a diferença entre usar uma tupla em vez de uma lista e vice-versa?**
- Por serem imutáveis, tuplas representam informações que **não devem ser modificadas**,
 - Exemplos: **vetores dos embeddings** e **classes morfosintáticas** retornadas por um *tagger*.
- Arranjo das tuplas é similar ao das listas, só não se usam os métodos para modifica-las.
 - Apenas o **count()** e o **index()** estão disponíveis.

DICIONÁRIOS

- São estruturas de dados bem úteis que permitem armazenar e recuperar informações por **pares chave-valor**.
- Arrays associativos
 - Analogia ao *hashing*
- De forma mais “simples”: é uma lista que podemos acessar seus elementos através de uma chave.

DICIONÁRIOS

```
>>> dic = {}
>>> dic['roney'] = 6
>>> dic['thiago'] = 12
>>> dic['alguém'] = 4
>>> print(dic)
{'roney': 6, 'thiago': 12, 'alguém': 4}
>>> dic['roney']
6
>>> dic['roney'] = 'seis'
>>> dic['roney']
'seis'
>>> |
```

- Os valores podem ser representados por **qualquer tipo de dados e quaisquer estrutura**: uma string, int, bool, lista, tupla, outro dicionário...

DICIONÁRIOS

○ Como percorrer os dicionários:

```
>>> print(dic.keys())
dict_keys(['c', 'b', 'a', 1.5, 4])
>>> print(dic.values())
dict_values([10, 20, 30, 40, 50])
>>> type(dic.keys())
<class 'dict_keys'>
>>> dic.keys[0]
Traceback (most recent call last):
  File "<pysHELL#48>", line 1, in <module>
    dic.keys[0]
TypeError: 'builtin_function_or_method' object is not subscriptable
>>> list(dic.keys())
['c', 'b', 'a', 1.5, 4]
>>> list(dic.items())
[('c', 10), ('b', 20), ('a', 30), (1.5, 40), (4, 50)]
>>> list(dic.values())
[10, 20, 30, 40, 50]
>>> dic
{'c': 10, 'b': 20, 'a': 30, 1.5: 40, 4: 50}
>>> for k, v in dic.items():
    print(k, 'tem como valor', v)
```

```
c tem como valor 10
b tem como valor 20
a tem como valor 30
1.5 tem como valor 40
4 tem como valor 50
>>> |
```

DICIONÁRIOS

○ Como percorrer os dicionários:

```
>>> print(dic.keys())
dict_keys(['c', 'b', 'a', 1.5, 4])
>>> print(dic.values())
dict_values([10, 20, 30, 40, 50])
>>> type(dic.keys())
<class 'dict_keys'>
>>> dic.keys[0]
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    dic.keys[0]
TypeError: 'builtin_function_or_method' object is not subscriptable
>>> list(dic.keys())
['c', 'b', 'a', 1.5, 4]
>>> list(dic.items())
[('c', 10), ('b', 20), ('a', 30), (1.5, 40), (4, 50)]
>>> list(dic.values())
[10, 20, 30, 40, 50]
>>> dic
{'c': 10, 'b': 20, 'a': 30, 1.5: 40, 4: 50}
>>> for k, v in dic.items():
    print(k, 'tem como valor', v)
```

```
c tem como valor 10
b tem como valor 20
a tem como valor 30
1.5 tem como valor 40
4 tem como valor 50
>>> |
```

Verifico as chaves do dicionário

Verifico os valores presentes no dicionário

O tipo retornado é próprio do dicionário, então não consigo manipular...

Dessa forma, transformo-o em uma lista

Aqui temos as informações completas, com chave-valor

Finalmente, usando a estrutura de controle **for**, podemos iterar pelo dicionário e retornar informações!

ARQUIVOS

- Arquivos são um tipo nativo do Python que permite a sua manipulação sem precisar importar nenhum módulo.
- Primeiro é necessário **criar um objeto que represente o arquivo** e então **abri-lo**, para começar a manipular:

```
1 infile = open("qbdata.txt", 'r')
2 outfile = open("result.txt", 'w')
```

ARQUIVOS

```
1 infile = open("qldata.txt", 'r')
2 outfile = open("result.txt", 'w')
```

- Importante: **atenção nos parâmetros** para abertura de arquivo.
 - **r** = *read*, ou seja, apenas para **leitura** do arquivo;
 - **w** = *write*, ou seja, apenas para **escrita** no arquivo;
 - **a** = *append*, ou seja, **escrita no final** do arquivo.
 - Existem outros parâmetros mais direcionados, mas que não serão explorados nessa aula.
- Ao final do uso do arquivo, sempre fechá-lo com o comando **close()**.

ARQUIVOS

- Assim que o arquivo é aberto, seu conteúdo é armazenado na variável criada para tal. A partir daí, a manipulação deve ser feita com essa variável.

```
4 conteudo = infile.read()
5 print(conteudo)
6
```

Colt McCoy QB, CLE	135	222	1576	6	9	60.8%	74.5
Josh Freeman QB, TB	291	474	3451	25	6	61.4%	95.9
Michael Vick QB, PHI	233	372	3018	21	6	62.6%	100.2
Matt Schaub QB, HOU	365	574	4370	24	12	63.6%	92.0
Philip Rivers QB, SD	357	541	4710	30	13	66.0%	101.8
Matt Hasselbeck QB, SEA	266	444	3001	12	17	59.9%	73.2
Jimmy Clausen QB, CAR	157	299	1558	3	9	52.5%	58.4
Joe Flacco QB, BAL	306	489	3622	25	10	62.6%	93.6
Kyle Orton QB, DEN	293	498	3653	20	9	58.8%	87.5

- Todo o conteúdo do arquivo em uma **única variável**

ARQUIVOS

- Assim que o arquivo é aberto, seu conteúdo é armazenado na variável criada para tal. A partir daí, a manipulação deve ser feita com essa variável.

```
4 lista_conteudo = infile.readlines()
5 print(lista_conteudo)
6
```

```
['Colt McCoy QB, CLE 135 222 1576 6 9 60.8% 74.5\n', 'Josh Freeman QB, TB
291 474 3451 25 6 61.4% 95.9\n', 'Michael Vick QB, PHI 233 372 3018 21
6 62.6% 100.2\n', 'Matt Schaub QB, HOU 365 574 4370 24 12 63.6% 92.0\n',
'Philip Rivers QB, SD 357 541 4710 30 13 66.0% 101.8\n', 'Matt Hasselbeck
QB, SEA 266 444 3001 12 17 59.9% 73.2\n', 'Jimmy Clausen QB, CAR 157 299
1558 3 9 52.5% 58.4\n', 'Joe Flacco QB, BAL 306 489 3622 25 10 62.6%
93.6\n', 'Kyle Orton QB, DEN 293 498 3653 20 9 58.8% 87.5\n', 'Jason
Campbell QB, OAK 194 329 2387 13 8 59.0% 84.5\n', 'Peyton Manning QB, IND
450 679 4700 33 17 66.3% 91.9\n', 'Drew Brees QB, NO 448 658 4620 33 22
68.1% 90.9\n', 'Matt Ryan QB, ATL 357 571 3705 28 9 62.5% 91.0\n', 'Matt
Cassel QB, KC 262 450 3116 27 7 58.2% 93.0\n', 'Mark Sanchez QB, NYJ 278
507 3291 17 13 54.8% 75.3\n', 'Brett Favre QB, MIN 217 358 2509 11 19
```

- Cada linha do arquivo em uma lista

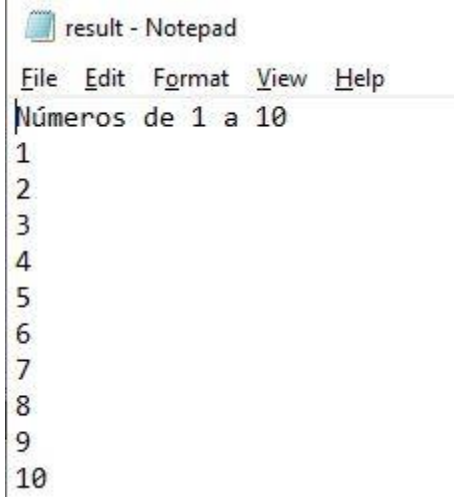
ARQUIVOS

○ Escrita:

A função **range()** retorna uma série numérica no intervalo enviado como argumento.

```
2 outfile = open("result.txt", 'w')
3
4 outfile.write("Números de 1 a 10\n")
5 for i in range(1,11):
6     outfile.write(str(i)+'\n')
7
```

[Finished in 0.4s]



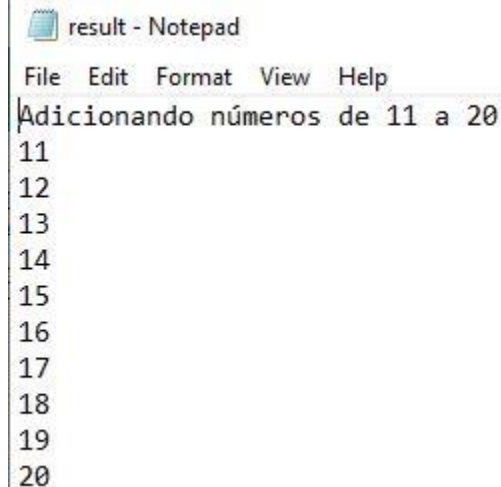
```
result - Notepad
File Edit Format View Help
Números de 1 a 10
1
2
3
4
5
6
7
8
9
10
```

ARQUIVOS

○ Escrita:

```
2 outfile = open("result.txt", 'w')
3
4 outfile.write("Adicionando números de 11 a 20\n")
5 for i in range(11,21):
6     outfile.write(str(i)+'\n')
7
```

[Finished in 0.4s]

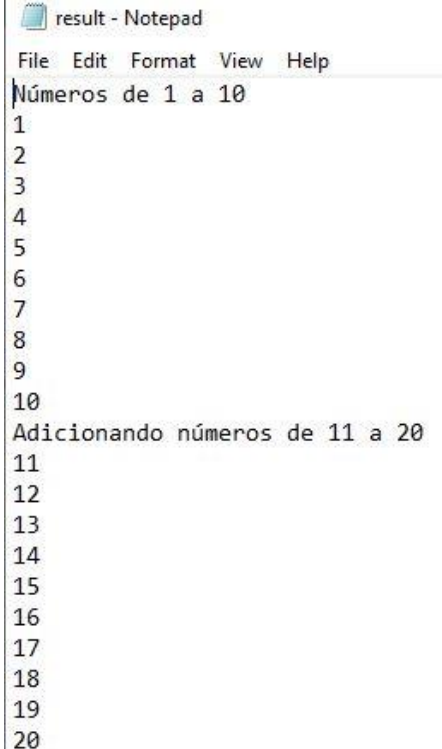


```
result - Notepad
File Edit Format View Help
Adicionando números de 11 a 20
11
12
13
14
15
16
17
18
19
20
```

ARQUIVOS

- Escrita com *append* (**a**):

```
2 outfile = open("result.txt", 'a')
3
4 outfile.write("Adicionando números de 11 a 20\n")
5 for i in range(11,21):
6     outfile.write(str(i)+'\n')
7
```



```
result - Notepad
File Edit Format View Help
Números de 1 a 10
1
2
3
4
5
6
7
8
9
10
Adicionando números de 11 a 20
11
12
13
14
15
16
17
18
19
20
```

ARQUIVOS

- Na escrita, a função `write()` aceita apenas **strings**, ou seja, se você colocar como parâmetro um inteiro ou qualquer outro tipo, um erro é lançado.
- Basta fazer o *cast* da variável ou da informação que pretende escrever no arquivo por meio de **`str(dado)`**.

```
2 outfile = open("result.txt", 'a')
3
4 outfile.write("Adicionando números de 11 a 20\n")
5 for i in range(11,21):
6     outfile.write(str(i)+'\n')
7
```

STRINGS

- Tipo de dados mais importante no trabalho em PLN, afinal, é o tipo que representa os **textos!**
- O Python define o tipo de dados como **str**
- As strings podem ser delimitadas por **aspas simples**, **aspas duplas** ou **aspas triplas**.

```
>>> 'aspas simples'
'aspas simples'
>>> "aspas duplas"
'aspas duplas'
>>> """aspas triplas"""
'aspas triplas'
>>> """aspas triplas tem uma propriedade que ignoram
a quebra de linha, então a string aparece
como ela é escrita"""
'aspas triplas tem uma propriedade que ignoram\na quebra de linha, então a string aparece\ncomo ela é escrita'
>>> |
```

STRINGS

- Interessante é que as strings são **imutáveis**, ou seja, da forma que elas são criadas, não é possível alterá-las **diretamente**. Exemplo:

```
>>> r = 'roney'
>>> r[0] = 'l'
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    r[0] = 'l'
TypeError: 'str' object does not support item assignment
```

- Entretanto, é possível **manipulá-las de diversas formas**. Se é necessário essa alteração, pode ser feito assim:

```
>>> nr = 'l' + r[1:]
>>> nr
'loney'
```


STRINGS

- Porém, é possível adicionar novas informações àquela string já criada por meio do operador +

```
>>> r = r + 'lira'
>>> r
'roneylira'
>>> |
```

- O operador + também serve para **concatenar strings com outros tipos** em string.

```
>>> pi = 3.14
>>> info = 'o valor de pi é = ' + pi
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    info = 'o valor de pi é = ' + pi
TypeError: can only concatenate str (not "float") to str
>>> info = 'o valor de pi é = ' + str(pi)
>>> info
'o valor de pi é = 3.14'
>>>
```

STRINGS

- E, claro, é possível usar o *slicing* na string.

```
>>> string = 'roney'  
>>> string[0]  
'r'  
>>> string[1:]  
'oney'  
>>> string[2:4]  
'ne'  
>>> string[:]  
'roney'  
>>> string[1:10]  
'oney'  
>>> |
```

STRINGS

- String possuem uma variedade ENORME de funções. Aqui as que serão trabalhadas nessa aula:
 - `len()`
 - `strip()`
 - `lower()`, `upper()`
 - `replace()`
 - `split()`
- Uma lista com mais funções sobre strings pode ser encontrada [aqui](#).

STRINGS

- **len()** – retorna o **tamanho** da string em referência a **caracteres e espaços**.

```
>>> s = 'instituto de ciências matemáticas e de computação'
>>> len(s)
49
>>> r = 'roney l'
>>> len(r)
7
```

- **lower()** e **upper()** – retorna a string **inteiramente** em **minúsculo** ou em **maiúsculo**.

```
>>> k = 'Roney Lira'
>>> k.lower()
'roney lira'
>>> k.upper()
'RONEY LIRA'
>>> |
```

STRINGS

- **strip()** – **exclui os espaços** existentes no início ou no final da string.

```
>>> r = ' Aula sobre strings '  
>>> r.strip()  
'Aula sobre strings'  
>>> |
```

- **replace()** – **substitui** uma string com outra string. Modo mais fácil de fazer a alteração na string que foi comentado um pouco antes.

```
>>> r = 'roney'  
>>> r.replace('r', 'l')  
'loney'  
>>> r = 'roney lira'  
>>> r.replace('r', 'l')  
'loney lila'  
>>> |
```

STRINGS

- **split()** – retorna um conjunto de **substrings** em **forma de lista**.
 - Podem ser usados parâmetros para fazer essa separação: uma palavra, um caractere, um escape, entre outros. Caso não seja utilizado **nenhum** argumento, a função trata como **separação por espaços**.

```
>>> s = 'instituto de ciências matemáticas e de computação'
>>> s.split()
['instituto', 'de', 'ciências', 'matemáticas', 'e', 'de', 'computação']
>>> s.split('matemáticas')
['instituto de ciências ', ' e de computação']
>>> s.split('de')
['instituto ', ' ciências matemáticas e ', ' computação']
>>> |
```

- Veja nesse exemplo que o `split` pode ser “gerador” de utilização de várias outras funções, como o `strip()`.

STRINGS

- **split()** – retorna um conjunto de **substrings** em **forma de lista**.
 - Interessante que agora podemos contar a **quantidade de palavras** que existem numa string.

```
>>> s = 'instituto de ciências matemáticas e de computação'
>>> len(s.split())
7
```

- A operação **inversa** do **split()** é o **join()**.

```
>>> sl = s.split()
>>> sl
['instituto', 'de', 'ciências', 'matemáticas', 'e', 'de', 'computação']
>>> sj = " ".join(sl)
>>> sj
'instituto de ciências matemáticas e de computação'
>>> sj = "/" .join(sl)
>>> sj
'instituto/de/ciências/matemáticas/e/de/computação'
>>> sj = "-".join(sl)
>>> sj
'instituto-de-ciências-matemáticas-e-de-computação'
>>>
```

STRINGS

- Caracteres de **escape**:
 - `\n` = quebra de linha
 - `\t` = tabulação
 - `\'` = aspas simples
 - `\\` = barra invertida
 - Entre outros...

```
1 s = 'instituto\n de ciências matemáticas\n e de computação'  
2 print(s)  
3 s = 'instituto de \'ciências matemáticas\' e de computação'  
4 print(s)  
5 s = 'instituto \t de ciências matemáticas e \t de computação'  
6 print(s)  
7 s = 'instituto \\ de ciências matemáticas e \\ de computação'  
8 print(s)
```

```
instituto  
de ciências matemáticas  
e de computação  
instituto de 'ciências matemáticas' e de computação  
instituto \t de ciências matemáticas e \t de computação  
instituto \\ de ciências matemáticas e \\ de computação
```


EXERCÍCIOS DE FIXAÇÃO

- 1. Dado o arquivo qbdata.txt, retorne o *rating* de cada QB na forma “nome do QB teve valor XX.X’ e escreva em um arquivo novo.
 - Arquivo qbdata.txt disponível [aqui](#).

EXERCÍCIOS DE FIXAÇÃO

- 1. Dado o arquivo `qbdata.txt`, retorne o rating de cada QB na forma “nome do QB teve valor `XX.X`”

```
1 |Colt McCoy QB, CLE 135 222 1576 6 9 60.8% 74.5
2 |Josh Freeman QB, TB 291 474 3451 25 6 61.4% 95.9
3 |Michael Vick QB, PHI 233 372 3018 21 6 62.6% 100.2
4 |Matt Schaub QB, HOU 365 574 4370 24 12 63.6% 92.0
5 |Philip Rivers QB, SD 357 541 4710 30 13 66.0% 101.8
6 |Matt Hasselbeck QB, SEA 266 444 3001 12 17 59.9% 73.2
7 |Jimmy Clausen QB, CAR 157 299 1558 3 9 52.5% 58.4
```

- Percebe-se que as informações que são requeridas são a primeira e a última de cada linha;
- É visto também que a separação entre as informações é feita por meio de espaços ou tabulação;
- Algoritmo: para cada linha, fazer a separação dela e por meio do *slicing* de listas, pegar a primeira e a última informação e montar a string final.

EXERCÍCIOS DE FIXAÇÃO

- 2. Pensando em uma agenda, construa um dicionário com informações do contato sendo: **CPF, nome, telefone e user no Twitter**. Ao final, imprima todos os contatos na forma

CPF: nome, telefone (user)

EXERCÍCIOS DE FIXAÇÃO

- 2. Pensando em uma agenda, construa um dicionário com informações do contato sendo: **CPF, nome, telefone e user no Twitter**. Ao final, imprima todos os contatos na forma

CPF: nome, telefone (user)

- De cara já dá para perceber que a chave vai ser o CPF, então na criação do dicionário, deve-se incluir na forma **dic[CPF] = ?**
- Temos mais de uma informação. O que seria interessante usar? Uma lista, uma tupla, outro dicionário?
- Ao final, para imprimir todos os contatos, há claramente a manipulação de strings.