SI-107 - Introdução à Ciência dos Dados com Python

Tópicos Especiais do Curso de Sistemas de Informação no IFMA Monte Castelo, 2019

Aula 2 - Visão Geral da Linguagem Python

Prof. Josenildo Silva (jcsilva@ifma.edu.br)

Nota 1: Este notebook é material complementar para disciplina Python para Ciencia dos Dados ministrada no curso de SI do IFMA Monte Castelo.

Nota 2: Este capítulo é baseado no Cap. 2 (Sec. 2.3) do livro “Python para Ciência dos Dados” de McKINNEY e em vários sites encontrados na internet. Para maiores detalhes; veja referencias no final.

Preliminares

Para começar, vamos discutir alguns aspectos básicos da linguagem: tipos, blocos, objetos

Tipos

Em python utilizamos a função `type()` para saber o tipo de uma variável

```
a = 1
```

```
type(a)
```

```
int
```

```
a = 1.0
```

```
type(a)
```

```
float
```

```
a = 'hello'
```

```
type(a)
```

```
str
```

```
a = None
```

```
type(a)
```

```
NoneType
```

```
a = 1.0
```

A conversão de tipos pode ser realizada de modo explícito com as funções str(), int(), float()

```
str(1) + ' abcde'
```

```
'1 abcde'
```

```
float('1')+0.8
```

```
1.8
```

Cuidado: A conversão de float para int pode perder informação!

```
int(1.99)
```

```
1
```

Bloco de Comandos e Comentários

O Python usa espaço em branco para definir os blocos de comandos. Os dois pontos (:) indicam o inicio de um bloco identado e todo código precisa serguir a mesma identação até o fim do bloco.

```
if 1 < 3:
    print('menor')
else:
    print('maior')
```

```
menor
```

Também não é necessário encerrar uma linha de comando com ponto-e-virgula

O sinal de hash (#) é utilizado como marcador de comentário.

```
# o simbolo de hash `#` inicia um comentário. O interpretador ignora a linha toda
```

Observe que não há saída ao executar a célula acima.

O comando `pass` indica que nada será executado. Serve para marcar um local em código onde um bloco deve existir, mas não há comandos.

```
pass
```

Objetos

Tudo é objeto em Python: números, strings, e até funções. Alguns objetos podem ser modificados, mas alguns são **imutáveis**. Um exemplo são as strings, que uma vez criadas, não podem ser modificadas.

No jupyter ou IPython, use para autocompletar com atributos e métodos para objetos

```
a = 'ana'
```

Escreva `a.` e presione 'TAB'. Um menu com todos os métodos é mostrado

```
a.capitalize() # na versão atual, o primeiro método mostrado é o capitalize()
```

```
'Ana'
```

Tipagem Fraca

```
a = 5
```

```
type(a)
```

```
int
```

Ao atribuir um novo valor, a variável passa a ter outro tipo.

```
a = 'foo'
```

```
type(a)
```

```
str
```

A função **`isinstance()`** verifica se uma variável é de um tipo indicado em uma tupla passada como parâmetro

```
isinstance(a,(int,str))
```

```
True
```

Apesar da tipagem fraca, Python evita conversão implícita de tipos

```
'5'+ 5 # gera erro de tipo
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-9e35a679fb6e> in <module>
----> 1 '5'+ 5 # gera erro de tipo

TypeError: can only concatenate str (not "int") to str
```

```
5 + '5' # também gera erro de tipo
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-1a91024124c5> in <module>
----> 1 5 + '5' # também gera erro de tipo

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Importação de módulos

Para importar um módulo:

```
import nome_do_modulo as apelido
```

Exemplo:

```
import numpy as np
```

Para utilizar uma variável ou função de um módulo importado use a notação de ponto (.) igual ao acesso de atributos e métodos de objetos.

```
nome_do_modulo.funcao()
nome_do_modulo.variavel
```

Alternativamente, pode-se importar apenas algumas funções ou variáveis

```
from nome_do_modulo import func_1, ...,var_1, func_j, var_2, ...
```

Doc ?

Use `?` para acesar o docstring de funções ou para inspecionar objetos.

```
a?
```

Comece a programar ou gere código com IA.

Comparador is e ==

O operador is testa se duas variáveis referenciam um mesmo objeto. Por outro lado, o comparador == testa apenas o conteúdo dos objetos referenciados pelas duas variáveis.

```
a = [1,2,3]
b = a
a is b
```

```
True
```

Retorna True pois os valores dos objetos referenciados por a e b possuem o mesmo conteúdo.

```
a == b
```

```
True
```

Se criarmos outro objeto para b, o resultado será diferente

```
b = [1,2,3]
a is b
```

```
False
```

A comparação a==b ainda retorna True, pois ambos os objetos, embora diferentes, possuem os mesmo valores.

```
a == b
```

```
True
```

Estruturas de Controle

A linguagem python apresenta controles de decisão (if) e dois laços (for e while).em python apresenta controles de decisão (if) e dois laços (for e while).

Decisão com IF

```
 if condição:
    # comando para primeira condição
 elif condição:
    # comando para segunda condição
 else:
    # comando para o caso default
```

Exemplo de uso de if

```
a=2
b=2
if a < b:
 print('menor')
 print('a')
elif a > b:
    print('maior')
else:
    print('iguais')
```

```
iguais
```

Laço For

```
 for value in coleção:
    # faça algo com o valor
```

Exemplo de uso do for

```
numeros = [1,2,3,4,5]
for n in numeros:
    print(n,end="  ") # o argumento end foi utilizado para substituir o default '\n' por espaço em branco " "
```

```
1  2  3  4  5
```

Laço While

```
 while condição:
```

```
    # faça alguma coisa
```

```
numeros = [1,2,3,4,5]
n=0
while n < len(numeros):
    n+=1
    print(n,end=' ')
print('fim')
```

```
1 2 3 4 5 fim
```

```
max_iter=1000
i=0
soma = 1
while i < max_iter:
    i+=1
    soma+=1/i
print(soma)
```

```
8.485470860550365
```

Os comandos `continue` e `break` alteram a execução de um laço.

```
numeros = [1,2,3,4,5]
n=0
while n < len(numeros):
    n+=1
    if n%2 == 0:
        continue # interrompe a iteração atual e recomeça pelo teste da variável de controle
    print(n,end=' ')
```

```
1 3 5
```

```
numeros = [1,2,3,4,5]
n=0
while n < len(numeros):
    n+=1
    if n==3:
        break # interrompe o laço completamente
    print(n,end=' ')
```

```
1 2
```

A função **range()** é utilizada frequentemente em conjunto com o **for**. `range()` retorna um iterador que gera uma sequencia de inteiros com espaçamento igual.

```
range(start, stop[, step])
```

```
list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
list(range(3,10,2))
```

```
[3, 5, 7, 9]
```

```
for n in range(10):
    print(n,end=" ")
```

```
0 1 2 3 4 5 6 7 8 9
```

Consulte o docstring com **range?** para maiores informações

Estruturas de Dados

O python oferece as seguintes estruturas de dados: tuplas, listas, conjuntos e dicionários.

Tuplas

Uma tupla é uma sequencia imutável de objetos.

O modo mais simples de criar uma tupla é informar os valores separados por vírgula

```
t = 1,2,3,4,5
```

```
t
```

```
(1, 2, 3, 4, 5)
```

Podemos criar tuplas compostas indicando a estrutura através de parêntesis

```
t= (1,2,3),(4,5)
```

```
t
```

```
((1, 2, 3), (4, 5))
```

É possível criar tuplas a partir de listas e iteradores utilizando a função tuple().

```
t= tuple(range(10))
t
```

```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Também podemos criar uma tupla a partir de uma string.

```
t = tuple('hello world')
t
```

```
('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')
```

Elementos são acessados com colchetes [].

```
t[1]
```

```
'e'
```

Tuplas são imutáveis. Por isso, os objetos referenciados em cada posição não podem ser trocados por outro.

```
t[0]='H' # gera erro já que tuplas são imutáveis
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-45-e9d3ad28b410> in <module>
----> 1 t[0]='H' # gera erro já que tuplas são imutáveis

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

O operador + concatena tuplas.

```
u='a','b','c'
v=t+u
v
```

```
('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', 'a', 'b', 'c')
```

Empacotamento é a atribuição de uma tupla para uma variável. Ao atribuir uma tupla para uma variável, chamamos de **desempacotamento**.

```
t= (1,2,3) # empacotar
x,y,z = t # desempactar
```

```
x
```

```
1
```

Com o conceito de tuplas, pode-se fazer a troca de valores entre duas variáveis sem precisar declarar uma terceira variável.

```
a,b=1,2
```

```
a,b
```

```
(1, 2)
```

```
b,a=a,b
```

```
a,b
```

```
(2, 1)
```

Outro conceito muito útil em Python é o **fatiamento** (*slicing*). Um fatiamento retorna uma subsequencia da estrutura indicados pela posição inicial e final entre colchetes.

```
t= ('Joao', 18, 'SI', 12, 45)
s = t[1:3]  # s será um slice com os elementos 1 e 2. Note que o primeiro elemento é 0.
s
```

```
(18, 'SI')
```

Listas

Listas são sequencias mutáveis de objetos. Ao contrário de tuplas, as listas podem ser modificadas após a sua criação.

```
a = [1,2,3,'Ana',None]
```

```
a
```

```
[1, 2, 3, 'Ana', None]
```

Uma lista pode ser criada a partir de uma tupla. Neste caso a lista pode ser modificada, mas a tupla continua imutável.

```
t = ('ana', 'bia', 'carlos')
```

```
b = list(t)
print(t)
```

```
('ana', 'bia', 'carlos')
```

```
b
```

```
['ana', 'bia', 'carlos']
```

```
b[1]='diego'
b
```

```
['ana', 'diego', 'carlos']
```

Os principais métodos do objeto list são os seguintes:

- `append()` insere um elemento no final da lista
- `insert()` insere um elemento em uma posição informada
- `pop()` remove um elemento de uma posição, ou o último
- `remove()` remove a primeira instância de um valor informado
- `sort()` ordena os elementos da lista

Perceba que `pop()` possui um parâmetro opcional de posição. Se não for informada uma posição, então o último elemento será removido.

Dada uma lista, a palavra reservada `in` verifica se um determinado elemento está contido na lista.

```
'diego' in b
```

```
True
```

O operador `+` concatena duas listas

```
c = ['paula', 'marcelo']
d = b + c +b
```

```
d
```

```
['ana', 'diego', 'carlos', 'paula', 'marcelo', 'ana', 'diego', 'carlos']
```

As listas também permitem *slicing* do mesmo modo que tuplas.

```
a = ['ana', 'diego', 'carlos', 'fred', 'joao']
a[2:4]
```

```
['carlos', 'fred']
```

Funções úteis para listas

Algumas funções úteis para listas são: `enumerate()`, `sorted()`, `zip()` e `reversed()`

A função `enumerate()` produz um gerador de tuplas no formato (int, value). Ela é útil para controlar o indice de um iterador. Podemos com essa função fazer um laço sem nos preocuparmos com o indice.

Desse modo, não é necessário fazer laços gerando um range com o tamanho da lista explicitamente:

```
names = ['Bob', 'Alice', 'Guido']
for i in range(len(names)):
    print(f'{i}:{names[i]}')
```

```
0:Bob
1:Alice
2:Guido
```

Ao invés disso, podemos fazer:

```
names = ['Bob', 'Alice', 'Guido']
for index, value in enumerate(names):
    print(f'{index}: {value}')
```

```
0: Bob
1: Alice
2: Guido
```

```
list(enumerate(names))
```

```
[(0, 'Bob'), (1, 'Alice'), (2, 'Guido')]
```

```
for i,n in enumerate(a):
    print (i,":",n)
```

```
0 : ana
1 : diego
2 : carlos
3 : fred
4 : joao
```

A função `sorted()` produz uma nova lista com os valores da lista original ordenados. A lista original não é modificada pela função sorted().

```
a = [8,3,7,1,0,2,6]
a_s = sorted(a)
print(a)
print(a_s)
```

```
[8, 3, 7, 1, 0, 2, 6]
[0, 1, 2, 3, 6, 7, 8]
```

A função `zip()` produz um iterador com elementos de duas listas distintas. Com este iterador, você pode construir uma lista, tupla ou um dicionário.

```
a = [8,3,7,1,0,2,6]
b = ['a','b','c','d','e','f','g']
c=zip(a,b)
list(c) # ou dict(c), tuple(c)
#print(list(enumerate(b)))
```

```
[(8, 'a'), (3, 'b'), (7, 'c'), (1, 'd'), (0, 'e'), (2, 'f'), (6, 'g')]
```

A função reversed() produz um iterador que percorre uma lista na ordem inversa.

```
b = ['a','b','c','d','e','f','g']
c = reversed(b)
list(c)
```

```
['g', 'f', 'e', 'd', 'c', 'b', 'a']
```

Conjuntos

Um conjunto é uma sequencia não ordenada de objetos únicos. Para criar um conjunto declaramos uma sequencia de elementos entre chaves {} e separados por vírgula.

```
c = {1,2,3,4,5}
c
```

```
{1, 2, 3, 4, 5}
```

```
type(c)
```

```
set
```

Alternativamente, pode-se criar um conjunto a partir de um iterador utilizando a função set()

```
set([6,7,8])
```

```
{6, 7, 8}
```

Atenção: Caso você queira criar um conjunto vazio, não use {}, pois este comando cria um dicionário vazio. Para criar um conjunto vazio utilize a função set() sem parâmetros.

```
d = {} # CUIDADO! Cria um dicionário, não um conjunto vazio!
```

```
type(d) # não é um conjunto!
```

```
dict
```

Python suporta as principais operações matemáticas sobre conjuntos tais como união, interseção, diferença, etc.

```
d = {3,5,8,9}
```

```
c.intersection(d)
```

```
{3, 5}
```

```
c.union(d)
```

```
{1, 2, 3, 4, 5, 8, 9}
```

```
c.difference(d)
```

```
{1, 2, 4}
```

Dicionários

Um dicionário é um conjunto de pares chave-valor. Internamente o dicionário é implementado com uma tabela hash aprimorada para melhorar a taxa de colisão e manter o tempo de busca.

Uma **chave** só pode ser um objeto imutável tais como int, float, ou string. Um **valor** pode ser qualquer objeto Python. Uma tupla pode servir como chave, mas seus elementos tem que ser todos imutáveis.

```
d = dict()
```

Também é possível criar um dicionário apenas utilizand a notação de chaves `{}`

```
d = {}
```

A função enumerate pode ser util para criar as chaves para os valores de uma lista.

```
a = ['Ana', 'Bia', 'Carla', 'Diane','Edson']
d = dict(enumerate((a)))
```

```
d
```

```
{0: 'Ana', 1: 'Bia', 2: 'Carla', 3: 'Diane', 4: 'Edson'}
```

Uma aplicação natural para os dicionário é a organização de informações em pares de atributos com seus valores. Considere as seguintes informações:

- Nome
- Idade
- Amigos

Um dicionário é um modo natural de organizar estas informações para cada pessoa.

```
pessoa = {'nome': 'Ana', 'idade':23, 'amigos':['Bia','Fred','Carlos']}
```

Cada atributo (chave) pode ser acessado com a notação de colchetes.

```
pessoa['nome']
```

```
'Ana'
```

Clique duas vezes (ou pressione "Enter") para editar

Utilizando a notação de colchetes podemos adicionar novos atributos dicionário

Utilizando a notação de colchetes podemos adicionar novos atributos ao dicionário

```
pessoa['conjuge']='Joao'
pessoa
```

```
{'nome': 'Ana',
 'idade': 23,
 'amigos': ['Bia', 'Fred', 'Carlos'],
 'conjuge': 'Joao'}
```

O operador del retira um par chave-valor do dicionário a partir de uma chave informada

```
del pessoa['conjuge']
pessoa
```

```
{'nome': 'Ana', 'idade': 23, 'amigos': ['Bia', 'Fred', 'Carlos']}
```

O método `pop()` tem o mesmo efeito, mas retorna o valor removido.

```
pessoa['conjuge']='Joao'
```

```
valor = pessoa.pop('conjuge')
valor
```

```
'Joao'
```

```
pessoa
```

```
{'nome': 'Ana', 'idade': 23, 'amigos': ['Bia', 'Fred', 'Carlos']}
```

O método `update()` modifica os valores de um dicionário existente, ou adiciona pares que ainda não existam. Vamos modificar a idade de Ana para 25 e acrescentar um atributo novo, salário, com seu respectivo valor.

```
pessoa.update({'salario':8000,'idade':25})
pessoa
```

```
{'nome': 'Ana',
 'idade': 25,
 'amigos': ['Bia', 'Fred', 'Carlos'],
 'salario': 8000}
```

Os métodos `keys()` e `values()` retornam iteradores com chaves e valores, respetivamente.

```
pessoa.keys()
```

```
dict_keys(['nome', 'idade', 'amigos', 'salario'])
```

```
pessoa.values()
```

```
dict_values(['Ana', 25, ['Bia', 'Fred', 'Carlos'], 8000])
```

A função `itens()` retorna uma lista de tuplas no formado (chave,valor)

```
pessoa.items()
```

```
dict_items([('nome', 'Ana'), ('idade', 25), ('amigos', ['Bia', 'Fred', 'Carlos']), ('salario', 8000)])
```

Um dicionário pode ser visto como uma lista de tuplas. Desse modo, pode ser criado a partir de uma combinação de listas com zip.

```
atributos = ['nome','idade','salario']
valores = ['Ana', 23, 8000]
```

```
p = dict (zip(atributos, valores))
p
```

```
{'nome': 'Ana', 'idade': 23, 'salario': 8000}
```

Abrangência (comprehension) de listas, dicionários e tuplas

Abrangência de coleções é uma maneira de construir uma nova coleção aplicando filtro e transformação em uma coleção original.

É equivalente a utilizar um laço for com um if para adicionar elementos na nova coleção.

A sintaxe para criar uma lista com abrangência é a seguinte:

```
[expr for val in collection if condition]
```

Por exemplo, imagine que temos uma lista de números

```
a=[1,2,-3,4,5,6,-7]
```

e queremos gerar uma lista com os quadrados dos seus valores. Podemos fazer isso com abrangência do seguinte modo:

```
print(a)
[x**2 for x in a]
```

```
[1, 2, -3, 4, 5, 6, -7]
[1, 4, 9, 16, 25, 36, 49]
```

Excluindo os números negativos, temos

```
print(a)
[x**2 for x in a if x >-1]
```

```
[1, 2, -3, 4, 5, 6, -7]
[1, 4, 16, 25, 36]
```

Exemplo de uso de comprehension

Vamos tentar um exemplo um pouco mais elaborado. Considere que temos registros sobre pessoas, armazenado em uma lista chamada dados.

```
p1 = {'nome': 'Ana', 'idade':23, 'amigos':['Bia','Fred','Carlos']}
p2 = {'nome': 'Bia', 'idade':27, 'amigos':['Ana','Davi']}
p3 = {'nome': 'Carlos', 'idade':26, 'amigos':['Davi','Bia']}
dados = [p1,p2,p3]
dados
```

```
[{'nome': 'Ana', 'idade': 23, 'amigos': ['Bia', 'Fred', 'Carlos']},
 {'nome': 'Bia', 'idade': 27, 'amigos': ['Ana', 'Davi']},
 {'nome': 'Carlos', 'idade': 26, 'amigos': ['Davi', 'Bia']}]
```

Como podemos fazer consultas a esta lista? Por exemplo, como consultar as pessoas com mais de 25 anos?

```
[p for p in dados if p['idade']>25]
```

```
[{'nome': 'Bia', 'idade': 27, 'amigos': ['Ana', 'Davi']},
 {'nome': 'Carlos', 'idade': 26, 'amigos': ['Davi', 'Bia']}]
```

Ou ainda, somente o nome das pessoas que atendam este critério

```
[p['nome'] for p in dados if p['idade']>25]
```

```
['Bia', 'Carlos']
```

Se preferir, utilize a função list() ao invés da notação de colchetes.

```
list(x for x in dados if x['idade']>25)
```

```
[{'nome': 'Bia', 'idade': 27, 'amigos': ['Ana', 'Davi']},
 {'nome': 'Carlos', 'idade': 26, 'amigos': ['Davi', 'Bia']}]
```

A estratégia de abrangencia para descrever como coleções são formadas pode ser utilizada para criação de dicionários.

Por exemplo, podemos criar um dicionário derivado, onde apenas alguns campos são utilizados.

```
p1 = {'nome': 'Ana', 'idade':23, 'amigos':['Bia','Fred','Carlos']}
#p1.items()
colunas = ['nome','idade']
{k:v for k,v in p1.items() if k in colunas}
```

```
{'nome': 'Ana', 'idade': 23}
```

Podemos aninhar a abrangência. Por exemplo, o código abaixo cria uma lista de pessoas a partir da lista `dados` mas apenas com nome e idade

```
[{k:v for k,v in p.items() if k in colunas} for p in dados if p['idade']>25]
```

```
[{'nome': 'Bia', 'idade': 27}, {'nome': 'Carlos', 'idade': 26}]
```

Funções

Uma função é um trecho de código nomeado que produz um resultado a partir de parâmetros dados. Para definir uma função, pode-se utilizar funções nomeadas ou funções anônimas.

Nas funções nomeadas o trecho de código recebe um nome para que possa ser invocado posteriormente em outro ponto do programa. Já as funções anônimas são utilizadas quando o código não tem previsão de ser reutilizado, portanto não é necessário nomear o trecho de código.

Em Python, e várias outras linguagens, funções anônimas são chamadas de *expressões lambda*.

Funções Nomeadas

Funções nomeadas são criadas com a palavra reservada def indicando o nome e os parâmetros da função. Não é necessário informar um tipo de retorno. A chamada de uma função é feita através do nome da função com parâmetros. Se o Python não encontrar um comando return um valor `None` será retornado automaticamente.

Sintaxe:

```
def nome_da_função (parâmetros):
    #codigo identado
```

Os parâmetros podem ser posicionais ou identificados (keywords).

Os **parâmetros identificados** por palavra chave na declaração da função podem ser omitidos na chamada da mesma. Também podem ser definidos com valores default. Os parâmetros identificados devem sempre ser declarados após os posicionais, se houver.

```
def media (x,y,tipo='aritmetica'):
    if tipo=='aritmetica':
       return (x+y)/2
    elif tipo=='harmonica':
       return (2*x*y)/(x+y)
```

Funções são **chamadas** por nome seguido de parentesis e seus parâmetros:

```
func(a,b)
```

```
media(2,5)
```

```
3.5
```

```
media(2,5,tipo='harmonica')
```

```
2.857142857142857
```

Na chamada de função, todos os parâmetros podem ser explícitos e dados em qualquer ordem. Isso ajuda a legibilidade do código.

```
media(tipo='harmonica',y=3,x=4)
```

```
3.4285714285714284
```

Lembre-se: os parâmetros posicionais não podem ser omitidos.

```
media(tipo='harmonica',y=3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-110-93bc20794b77> in <module>
----> 1 media(tipo='harmonica',y=3)

TypeError: media() missing 1 required positional argument: 'x'
```

Uma função em Python pode retornar uma tupla. Isso permite retornar multiplos valores de uma só vez. Para tanto, na chamada da função você deve desempacotar o retorno para várias variáveis.

```
def descreva (numeros):
    minimo= min(numeros)
    maximo = max(numeros)
    med = sum(numeros)/len(numeros)
    return minimo,maximo,med
```

```
#descreva([11,9,7.5])
maximo,minimo,medio=descreva([11,9,7.5])


maximo, minimo, medio
```

```
(7.5, 11, 9.166666666666666)
```

Outra alternativa é retornar um dicionário. Em geral, esta alternativa torna o retorno mais legível.

```
def descreva (numeros):
    minimo= min(numeros)
    maximo = max(numeros)
    med = sum(numeros)/len(numeros)
    return {'minimo':minimo,'maximo': maximo,'media':med}


r=descreva([11,9, 7.5])
r
```

```
{'minimo': 7.5, 'maximo': 11, 'media': 9.166666666666666}
```

A possibilidade de retornar tuplas e dicionários dá grande flexibilidade para o desenvolvedor, uma vez que ele não precisa definir estruturas temporárias apenas para retornar valores de uma dada função.

Um fato importante sobre funções é que elas são objetos em Python. Desse modo, podem ser passadas como parâmetro, podem ser colocadas em listas, etc.

```
def media(x,y):
    return (x+y)/2


def desvio (x,y):
    return abs(x-y)/2**(1/2)


operacoes = [media,desvio]


operacoes
```

```
[<function __main__.media(x, y)>, <function __main__.desvio(x, y)>]
```

```
for op in operacoes:
    print(op(3,5))

[op(3,5) for op in operacoes]
```

```
4.0
1.414213562373095
[4.0, 1.414213562373095]
```

O atributo `__name__` é uma string com o nome de uma função. Com ele, a saida fica mais legível. Note que transformamos float em string para concatenarmos o nome das funções.

```
for op in operacoes:
    print(op.__name__+": "+str(op(3,5)))
```

```
media: 4.0
desvio: 1.414213562373095
```

Funções anonimas (lambda)

Em alguns contextos, não precisamos criar um código reutilizável. Nestes casos, utilizamos as funções anônimas. Em Python, utililzamos uma **expressão lambda** para criar uma função anônima.

Uma função lambda possui apenas um comando e produz um único valor de retorno. Sua sintaxe é a seguinte:

```
lambda argumentos: expressão
```

Por exemplo, podemos definir uma expressão para calcular o quadrado de um número.

```
lambda x: x**2
```

Funções lambdas são geralmente utilizadas em combinação com outras funções tais como a função map(). A função map() aplica a expressão lambda nos elementos de uma coleção e retorna um iterador com os resultados. O iterador pode ser utilizado em uma função list(), set(), etc.

Exemplo: Vamos aplicar uma função lambda que receba um parametro x e devolva x elevado ao quadrado. Essa função é aplicada sobre os elementos da coleção nums e o iterador it_quads é retornado. O iterador é utilizado para gerar uma lista cujos elementos são o quadrado dos números da lista original.

```
nums =[1,2,3,4,5]

it_quads = map(lambda x: x**2,nums)

list(it_quads)
```

```
[1, 4, 9, 16, 25]
```

Variáveis e passagem de parâmetros

Atribuição de estruturas de dados em Python cria apenas uma referencias e não uma cópia dos valores dos objetos.

```
a = [2] # cria uma lista com um único elemento, o número 2
b = a
```

Após a atribuição b é uma segunda referencia para lista [2], além de a. Se modificarmos o valor da lista originalmente referenciada por a, a referencia b também apresentará a modificação.

```
b # antes
```

```
[2]
```

```
a.append(4)

b # depois
```

```
[2, 4]
```

Quando passamos parâmetros para funções, variáveis locais são criadas para referenciar os objetos sem copiá-los.

Conclusão

Isto encerra nossa pequena introdução aos principais aspectos da linguagem Python. Muitos detalhes foram simplificados para que este material não ficasse muito extenso. Caso você deseje se aprofundar mais em algum dos tópicos, use as referências fornecidas abaixo. Nas próximas aulas iremos estudar bibliotecas de apoio à análise de dados: numpy e pandas.

Referências

1. MENEZES, Nilo N. C. “Introdução à programação com Python”. Novatec, 2019.
2. McKENNEY, Wes (2017). “Python for Data Analysis”. O’Reilly, 2nd Edition.
3. Python Software Foundation (2019). “The Python Tutorial”. https://docs.python.org/3/tutorial/
4. REITER, R. “Iteractive Tutorials: Python”. https://www.learnpython.org
5. Data Camp. “Introduction to Python”. https://www.datacamp.com/courses/intro-to-python-for-data-science
6. Fernando Masanori. “Python para Zumbis”. https://www.pycursos.com/python-para-zumbis/
7. CCSL do IME/USP. “Introdução à Ciência da Computação com Python”. Série de videos. https://www.youtube.com/playlist?list=PLcoJJSvnDgcKpOi_UeneTNTIVOigRQwcn
8. Pyhton Cheat Sheet. https://www.pythoncheatsheet.org/

9. Dev Media. “Python Tutorial”. https://www.devmedia.com.br/python-tutorial/33274
10. DBAder. "Python Enumerate". https://dbader.org/blog/python-enumerate