Universidade Federal Fluminense

Introdução ao Python

Autores:

Bernardo Corrêa Gonçalves
Estudante de Engenharia Elétrica

Gustavo Caetano Marçal
Estudante de Engenharia Elétrica

Maurício Raphael Waisblum Barg
Estudante de Engenharia Elétrica

Tayná Ferreira Santos
Estudante de Engenharia Elétrica

Sumário

1. Introdução

O Python é uma linguagem de programação **interpretada, orientada a objetos**. Ela se tornou extremamente popular devido a sua sintaxe simples e sua fácil leitura, além de sua excelente portabilidade. Por ser popular, o Python já conta com inúmeros *frameworks* e bibliotecas muito úteis, tornando-o uma linguagem muito poderosa.

1.1. Sintaxe

As variáveis no Python, diferente de outras linguagens de programação, não precisam ter seu tipo declarado. Ou seja, não é necessário informar previamente ao programa se a variável recebe um valor inteiro, real, lógico, etc, basta inicializar a variável direto com o valor.

```
In [1]: abc = 'hello world'

In [2]: abc
Out[2]: 'hello world'

In [3]: abc = 10

In [4]: abc
Out[4]: 10
```

1.2. Indentação

Quando escrevemos um programa em Python, devemos obrigatoriamente indentá-lo corretamente. Enquanto que em outras linguagens procuramos indentar o código para deixá-lo mais legível, no Python somos forçados a fazê-la, uma vez que o programa não funcionará sem a indentação.

```
In [5]: def mostrar_indentacao(x):
   ...:     for numero in range(x):
   ...:         if numero % 2 == 0:
   ...:             print numero
   ...:
```

1.3. Comentários

Comentários são trechos de código que não serão interpretados. Eles servem para comentar o que cada trecho de código faz, para que outros possam entender. Em Python, os comentários são feitos da seguinte forma:

‘#’ : Para comentários em linha única

‘ “”” (texto) “”” ‘ : Para blocos de texto.

```
# -*- coding: utf-8 -*-
# Posso escrever o que quiser que não fará diferença ao meu programa

""" assim como aqui também não
pois posso comentar até aonde
eu queira"""
```

1.4. Operações

Operações aritméticas simples podem ser feitas no Python de forma natural, usando os sinais que estamos acostumados (+, -, /, *). Além disso, tem-se o ‘%’ e o ‘**’ que representam o resto da divisão e potenciação, respectivamente. A prioridade de operações é a natural da matemática. Porém, quanto mais parênteses forem utilizados é melhor para que não sejam cometidos erros.

```
In [7]: 1+1
Out[7]: 2

In [8]: 10*192
Out[8]: 1920

In [9]: 9/2
Out[9]: 4

In [10]: (10*3)*4/5
Out[10]: 24

In [11]: 9%4
Out[11]: 1

In [12]: 2**3
Out[12]: 8
```

2. Tipos básicos de variáveis

Dentre os vários tipos de variáveis que existem no Python, existem 4 principais tipos que ajudam a formar todas as outras. São eles: int, float, bool e string.

2.1. Int

Int são as variáveis que guardam valores inteiros. Com as próprias variáveis podem ser feitas as operações aritméticas simples já citadas, porém, se todas as variáveis forem do tipo int, a resposta será também um tipo int.

```
In [1]: a = 2

In [2]: b = 5

In [3]: a * b
Out[3]: 10

In [4]: b/a
Out[4]: 2
```

2.2. Float

As variáveis do tipo float guardam valores com casas decimais, ao contrário do tipo int. Porém, as mesmas contas podem ser feitas inclusive entre os dois tipos. A prioridade de resposta é sempre do tipo float. Lembrando que no lugar de “,” deve – se usar “.”.

```
In [5]: a = 2

In [6]: b = 5.0

In [7]: a * b
Out[7]: 10.0

In [8]: b/a
Out[8]: 2.5
```

2.3. Bool

São tipos de variáveis que guardam valores lógicos como verdadeiro e falso (no Python, “True” e “False”).

```
In [9]: a = True

In [10]: b = False

In [11]: b
Out[11]: False
```

Também são realizadas verificações a outras variáveis, como por exemplo, “será que determinada variável tem o valor 2?”. Esse tipo de pergunta é feita com o uso do “ == “. A resposta sempre será do tipo bool ou booleano.

```
In [12]: a = 2

In [13]: a ==2
Out[13]: True
```

2.3.1. Operações com booleanos

As operações com booleanos são um pouco diferentes das operações com outras variáveis. Neste caso, são feitas operações lógicas com o uso dos comandos “and”, “or” e “not”. A lógica é a mesma da tabela abaixo.

A	B	A e B	A ou B	nao A
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

```
In [14]: a = True

In [15]: b = False

In [16]: a and b
Out[16]: False

In [17]: a or b
Out[17]: True

In [18]: not a
Out[18]: False
```

2.4.String

String é um tipo de variável que lida com uma sequência de caracteres, sejam números ou letras. Para definir uma string basta associar uma variável a qualquer sequência de caracteres entre aspas duplas (“), ou simples (‘).

```
In [19]: a = "Lilly"

In [20]: a
Out[20]: 'Lilly'
```

Ao definir uma string utilizando aspas simples é preciso ter cuidado ao fazer uso de outra aspas simples dentro da mesma string, pois o programa interpretará realmente apenas o que estiver no primeiro par de aspas. Caso queira corrigir o problema deve-se utilizar as aspas duplas ou acrescentar uma barra invertida (\) antes das aspas.

```
In [21]: a = 'Essa foi a gota d'agua!'
  File "<ipython-input-21-3c1700090e85>", line 1
    a = 'Essa foi a gota d'agua!'
                          ^
SyntaxError: invalid syntax


In [22]: a = 'Essa foi a gota d\'agua!'

In [23]: a
Out[23]: "Essa foi a gota d'agua!"
```

2.4.1. Índices

Toda string é indexada, ou seja, todos os caracteres contidos numa string possuem uma posição, o primeiro caractere tem índice 0, o segundo tem índice 1, e assim por diante. Para chamar os caracteres de uma string pelo seu índice basta acrescentar seu índice entre colchetes.

```
In [23]: a
Out[23]: "Essa foi a gota d'agua!"

In [24]: a[2]
Out[24]: 's'
```

2.4.2. Operações com strings

Apesar de parecer estranho, algumas operações com strings podem ser feitas com o “+” (também chamada de concatenação), como também com o “*”.

```
In [25]: a = "Apostila"

In [26]: b = " de"

In [27]: c = " Python"

In [28]: a+b+c
Out[28]: 'Apostila de Python'

In [29]: c*3
Out[29]: ' Python Python Python'
```

2.4.3. Métodos

LEN (): Retorna o tamanho da string

.LOWER(): Retorna a string em caixa baixa

.UPPER(): Retorna a string em caixa alta

STR() : Transforma uma variável qualquer em uma string

```
In [8]: a = "Lilly"

In [9]: b = 10

In [10]: len(a)
Out[10]: 5

In [11]: b = str(b)

In [12]: type(b)
Out[12]: str

In [13]: a.lower()
Out[13]: 'lilly'

In [14]: a.upper()
Out[14]: 'LILLY'
```

2.5. Como imprimir variáveis dentro de uma string

No Python há uma excelente ferramenta de formatação quando se utiliza o print para mostrar alguma informação na tela com variáveis que é o uso dos seguintes caracteres: %s, %d, %f. Esses caracteres representam, respectivamente, uma string, um inteiro e um float.

```
In [30]: cachorro = "Gaya"

In [31]: racao = 1.5

In [32]: tempo = 5

In [33]: print "%s comeu %f kg de racao em apenas %d semanas!" %(cachorro, racao, tempo)
Gaya comeu 1.500000 kg de racao em apenas 5 semanas!
```

3. Entrada e saída de dados

Em muitos casos precisa-se que o programa receba e manipule dados de um usuário. No Python existem duas maneiras de ler dados, seja utilizando o comando raw_input() ou input(). No raw_input tudo o que for escrito pelo usuário será transformado em uma string. O input só aceita entrada de dados numéricos, que serão transformados em algum tipo já conhecido de variável. Para escrever uma string com o comando input é necessário o uso das aspas, o que não é necessário no caso do raw_input.

```
In [11]: a = raw_input("Digite o que voce quiser: ")

Digite o que voce quiser: ok, escreverei tudo que puder [1,2,3] 4

In [12]: a
Out[12]: 'ok, escreverei tudo que puder [1,2,3] 4'

In [13]: a = input("Digite o que voce quiser: ")

Digite o que voce quiser: 89

In [14]: type(a)
Out[14]: int
```

Para imprimir os dados é utilizado o comando print, conforme demonstrado a seguir:

>>> a = 12
>>> **print** a
>>> 12
>>> **print** “Minicurso de Python”
>>> Minicurso de Python

4. Listas e Dicionários

4.1. Lista

É um conjunto de valores ordenados, onde cada elemento é identificado por um índice, sendo que na linguagem Python o primeiro elemento é identificado pelo índice 0, o segundo pelo índice 1 e assim sucessivamente. Os elementos podem ser do tipo int, float ou até mesmo string.

Dentre as diversas formas de se criar uma lista, a mais simples delas é envolver seus elementos pelo uso de colchetes.

```
In [1]: lista1 = []

In [2]: lista2 = [1,2,3,4]

In [3]: lista3 = ["Hello", "Pet"]
```

OBS: No exemplo, a variável lista2 é extremamente semelhante a um vetor. Lista é a forma mais simples de representar um vetor, e

também pode ser usado para representar uma matriz. Porém, existem formas melhores, como a biblioteca “Numpy”.

4.1.1. Como chamar os elementos da lista por seus índices

Para acessar um determinado elemento de uma lista segue-se o mesmo princípio de uma string. Se existem n elementos, as posições deles serão definidas de 0 a n-1. Portanto, para acessar um elemento q, onde q<n, deve-se fazer o seguinte comando: nome_da_lista[q-1].

```
In [4]: lista = ["Como", "Python", "e", "divertido"]

In [5]: print lista[2]
e

In [6]: print lista[1]
Python
```

4.1.2. Operações com listas

As mesmas operações com strings podem ser feitas com lista. Então pode-se somar listas ou até mesmo multiplicá-las por um inteiro.

```
In [7]: lista2*3
Out[7]: [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]

In [8]: lista2 + lista3
Out[8]: [1, 2, 3, 4, 'Hello', 'Pet']
```

4.1.3. Métodos

.APPEND(): Adiciona determinado elemento à lista citada.

```
In [9]: a = [1,2,3]

In [10]: b = "Hello Pet"

In [11]: a.append(b)

In [12]: a
Out[12]: [1, 2, 3, 'Hello Pet']
```

.INDEX(): Exibe o índice da primeira ocorrência de determinado elemento.

```
In [13]: a.index('Hello Pet')
Out[13]: 3
```

.INSERT(): Insere um elemento em um índice escolhido pelo programador.

```
In [14]: a
Out[14]: [1, 2, 3, 'Hello Pet']

In [15]: a.insert(2,2015)

In [16]: a
Out[16]: [1, 2, 2015, 3, 'Hello Pet']
```

DEL: Retira o elemento de um índice escolhido pelo programador.

```
In [17]: a
Out[17]: [1, 2, 2015, 3, 'Hello Pet']

In [18]: del a[2]

In [19]: a
Out[19]: [1, 2, 3, 'Hello Pet']
```

.SORT(): Ordena determinada lista.

```
In [20]: b = [4,2,6,3,8]

In [21]: b.sort()

In [22]: b
Out[22]: [2, 3, 4, 6, 8]
```

.REMOVE(): Remove a primeira ocorrência de determinado elemento.

```
In [22]: b
Out[22]: [2, 3, 4, 6, 8]

In [23]: b.remove(6)

In [24]: b
Out[24]: [2, 3, 4, 8]
```

POP: Remove e retorna o último valor da lista. Também pode-se atribuir um índice para remoção, sendo retornado este valor removido.

```
In [38]: a = [1,2,3,5,4]

In [39]: print a
[1, 2, 3, 5, 4]

In [40]: a.pop()
Out[40]: 4

In [41]: print a
[1, 2, 3, 5]

In [42]: a.pop(0)
Out[42]: 1

In [43]: print a
[2, 3, 5]
```

4.2. Dicionário

Dicionários são estruturas de dados formados por pares de valores onde o primeiro valor é chamado de chave e o segundo de conteúdo. De certo modo eles são parecidos com listas, onde cada índice da lista pode ser representado agora por strings, além de inteiros e floats. Uma das maneiras mais fáceis de introduzir o dicionário é colocando seus elementos entre chaves da seguinte forma: {“chave”: “conteúdo”}

```
In [1]: a = {}

In [2]: b = {"joao":33631456, "Fabricio":24662132}
```

4.2.1. Acessando e criando chaves e conteúdos

Pode-se também acessar os conteúdos como na lista apenas colocando no lugar do que era o índice uma chave. Além disso, pode-se criar novas chaves apenas atribuindo valores.

```
In [3]: b["joao"]
Out[3]: 33631456

In [4]: b["Brenno"] = 21543256

In [5]: b
Out[5]: {'Brenno': 21543256, 'Fabricio': 24662132, 'joao': 33631456}
```

4.2.2. Métodos

.CLEAR(): Remove todos os elementos do dicionário.

```
In [5]: b
Out[5]: {'Brenno': 21543256, 'Fabricio': 24662132, 'joao': 33631456}

In [6]: b.clear()

In [7]: b
Out[7]: {}
```

.FROMKEYS(lista,valor): retorna um novo dicionário cujas chaves são os elementos de uma lista e cujos valores são iguais ao valor.

```
In [8]: {}.fromkeys(["Joao","Maria"],"irmao")
Out[8]: {'Joao': 'irmao', 'Maria': 'irmao'}
```

.GET(chave,valor): Obtém o conteúdo da chave. Se a chave não existir retorna o valor.

```
In [14]: dic.get("Brenno", "Nao existe")
Out[14]: 21543256

In [15]: dic.get("Caio", "Nao existe")
Out[15]: 'Nao existe'
```

.ITEMS(): Devolve uma lista com os pares chaves/valor, sem ordem específica.

```
In [16]: dic.items()
Out[16]: [('Brenno', 21543256), ('joao', 33631456), ('Fabricio', 24662132)]
```

.KEYS(): Devolve uma lista com as chaves, sem ordem específica.

```
In [17]: dic.keys()
Out[17]: ['Brenno', 'joao', 'Fabricio']
```

.VALUES():Devolve uma listas com os valores, sem ordem específica.

```
In [18]: dic.values()
Out[18]: [21543256, 33631456, 24662132]
```

.POP(chave): Devolve o valor correspondente a chave e remove o par chave/valor.

```
In [20]: dic.pop("joao")
Out[20]: 33631456

In [21]: dic
Out[21]: {'Brenno': 21543256, 'Fabricio': 24662132}
```

5. Controle de Fluxo

Em toda linguagem de programação estão presentes os controladores de fluxo, utilizados para que o programa execute determinadas instruções a partir de uma condição.

5.1.If, Elif e Else

No Python, o controle de fluxo que utiliza de condições para avaliar qual será o próximo comando a ser executado é o if. Com esse comando o interpretador é capaz de avaliar se uma condição é verdadeira e com isso realizar uma sequência de diferentes processos. O comando else é realizado caso não atenda a condição do if. Para que o programa seja capaz de avaliar condições é preciso utilizar comparadores lógicos, tais como:

== - IGUAL

!= - DIFERENTE

< - MENOR QUE

> - MAIOR QUE

<= - MENOR OU IGUAL

>= - MAIOR OU IGUAL

```
In [1]: numero = 2

In [2]: if numero%2 == 0:
   ...:     print "Esse numero e par"
   ...: else:
   ...:     print "Esse numero e impar"
   ...:
Esse numero e par


In [3]: numero = 3

In [4]: if numero%2 == 0:
   ...:     print "Esse numero e par"
   ...: else:
   ...:     print "Esse numero e impar"
   ...:
Esse numero e impar
```

Nesse programa ele avalia no caso if se o resto da divisão do número for 0. Em caso afirmativo ele irá mostrar na tela a frase “Esse numero e par”, caso contrário irá mostrar na tela a frase “ Esse numero e impar”.

OBS: Após os usos dos comandos if, elif e else é necessário o uso de dois pontos (:), e todo o código a ser seguido deve estar devidamente indentado.

5.1.1. Diferença entre elif e o else

O comando elif é uma abreviação de else if. O else complementa o if, ou seja, caso a condição não seja verdadeira os comandos a serem executados serão os que estiverem indentados ao else. Já o elif realiza os comandos seguintes avaliando uma nova condição.

OBS: Diferentemente de muitas linguagens, no Python o elif NUNCA estará dentro do if. A prioridade será totalmente do que vier antes.

```
In [6]: a = 5

In [7]: if a< 10:
   ...:     print "a<10"
   ...: elif a < 7:
   ...:     print "a<7"
   ...: else:
   ...:     pass
   ...:
a<10
```

6. Controle de Repetição

Os comandos de repetição são aplicados quando se deseja que o código seja realizado determinada quantidade de vezes ao longo do programa, originando um laço ou loop.

6.1.For

O comando for no Python é um pouco diferente de outras linguagens. No Python ele percorre elementos de um variável. Por exemplo, poderia percorrer os elementos de uma lista, as letras de uma string e etc.

Para usar o comando for será criado uma variável auxiliar, com um nome qualquer, de tal forma que a cada interação ela irá ser igualada a um elemento daquele objeto na ordem.

```
In [21]: lista = ['a', 'hello pet eletrica uff', 437]

In [22]: for i in lista:
    ...:     print i
    ...:
a
hello pet eletrica uff
437
```

Porém, também é possível usar o for da mesma forma que em outras linguagens, onde são feitos alguns comandos por uma determinada quantidade de vezes que seja necessária. Para esse fim pode-se usar a função range() que retorna uma lista com a quantidade de elementos que estiver no seu argumento.

```
In [23]: range(10)
Out[23]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Se esse comando for integrado ao for acontecerá o seguinte:

```
In [24]: a = 1

In [25]: for i in range(10):
    ...:     a = a*2
    ...:     print a
    ...:
2
4
8
16
32
64
128
256
512
1024
```

Ao ser percorrida uma sequencia pode-se obter o índice da posição atual e seu valor correspondente através da função **enumerate()**.

>>> **for** ordem, i **in enumerate**([“Arroz”,”Feijão”,”Carne”]):
>>> **print** ordem, i
>>> 0 Arroz
>>> 1 Feijão
>>> 2 Carne

6.1.1. For ... Else

Assim como no comando if, o for também possui a condição else. Essa última será executada apenas uma vez, assim que terminar a condição do for.

```
In [26]: a = 2

In [27]: for i in range(2):
    ...:     print a
    ...: else:
    ...:     print "acabou"
    ...:
2
2
acabou
```

6.2. While

O while é uma estrutura de repetição que executa um código sempre que determinada condição for verdadeira. A partir do momento que essa condição se tornar falsa, o programa prosseguirá sem acessar as informações do while novamente.

```
In [28]: condition = True

In [29]: while condition:
    ...:     print "Yeah!"
    ...:     condition = False
    ...:
Yeah!

In [30]: count = 0

In [31]: while count <5:
    ...:     print count
    ...:     count = count + 1
    ...:
0
1
2
3
4
```

OBS: Deve-se tomar muito cuidado para que em alguma hora a condição do while se torne falsa. Se isso nunca acontecer o programa irá executar o comando infinitamente. Uma maneira bastante útil de evitar esse tipo de desastre é o comando “break”. Ele faz com que o programa saia do laço automaticamente.

```
In [32]: count = 1

In [33]: a = 2

In [34]: while count == 1:
    ...:     print "Uhul!!"
    ...:     if a >4:
    ...:         break
    ...:     a =a +1
    ...:
Uhul!!
Uhul!!
Uhul!!
Uhul!!
```

6.2.1. While ... Else

Da mesma forma que no for e no if, pode ser colocado a condição else no comando while, sendo executado uma única vez assim que sair da condição imposta pelo comando.

```
In [35]: count = 0

In [36]: while count <5:
    ...:     print count
    ...:     count = count + 1
    ...: else:
    ...:     print "acabou"
    ...:
0
1
2
3
4
acabou
```

7. Funções

Em geral, quando fazemos um programa queremos utilizá-lo para realizar algumas ações muitas vezes fazendo com que o código fique longo e de difícil compreensão. Uma forma de tornar o código mais simples e organizado é substituindo ações repetidas por funções.

7.1.Sintaxe

No Python para definir funções basta usar o comando def nome_da_função(parâmetro1, parâmetro2, ...):

```
In [38]: def cubo(x):
    ...:     return x**3
    ...:

In [39]: n = input("Qual numero voce deseja elevar ao cubo?")

Qual numero voce deseja elevar ao cubo?3

In [40]: print cubo(n)
27
```

Quando definimos uma função no Python precisamos definir também um ou mais parâmetros, que são as variáveis utilizadas pela função. Funções que têm a necessidade de retornar um número ou uma string utilizam o comando return.

Para chamar uma função já definida no seu programa basta escrever o nome dela e entre parênteses indicar qual variável que a função tomará como parâmentro. Se sua função exigir mais de um parâmentro é importante a ordem dele quando for chamar a função.

7.2. Funções já definidas no Python

MAX(): Retorna o valor máximo de uma lista, dicionário ou string.

```
In [43]: a = [1,2,3,4,5]

In [44]: max(a)
Out[44]: 5
```

MIN():Retorna o valor mínimo de uma lista, dicionário ou string.

```
In [45]: a = [1,2,3,4,5]

In [46]: min(a)
Out[46]: 1
```

ABS(): Retorna o valor, em módulo, de um número.

```
In [47]: abs(-20)
Out[47]: 20
```

TYPE(): Retorna o tipo de qualquer objeto.

```
In [48]: type(a)
Out[48]: list
```

8. Classes

Classe é a estrutura básica do paradigma de orientação a objetos, é um tipo criado pelo usuário e representa conjuntos de objetos, abstrações

computacionais que atuam como entidades com características (atributos) e ações (métodos) semelhantes.

O uso das classes é destinado para resolver a questão de criação de objetos, ou seja, ao invés de criar objetos isolados, com definição de atributos e métodos para cada um, são aplicadas as classes para definir os atributos e métodos comuns a todos os objetos.

8.1.Sintaxe

A classe é criada em Python por meio da palavra reservada **class**. A sintaxe é a seguinte:

>>>class Nome_da_classe:

>>> <comandos>

No caso da classe não fazer nada é usada a palavra reservada **pass:**

>>>class Nome_da_classe:

>>> **pass**

É recomendado que se utilize letra maiúscula para iniciar o nome da classe.

8.2.Objetos

De modo geral as instâncias são usadas para a execução de um programa e não as classes. Instâncias podem ser vistas como indivíduos de uma classe, possuindo seus atributos. A sintaxe utilizada para produzir instância é a seguinte:

>>>Nome_da_classe(parâmetros)

Para referenciar uma dada instância criada lhe é atribuída uma variável, representando o objeto.

>>>variável = Nome_da_classe(parâmetros)

8.3. Construtor

Quando um objeto é criado, o construtor de classe é executado. Em Python, o construtor é um método especial chamado __new__(), após esta chamada, o método __init__() é chamado para inicializar uma nova instância. Quando um objeto é apagado o método __done__() é chamado.

```
>>>class Nome_da_classe:
>>>    def __init__(self, argumentos):
>>>        self.argumentos=argumentos
```

O variável self, que representa o objeto que precisa ser passado de forma explícita, é uma convenção que pode ser substituída por outra palavra, mas é uma boa prática mantê-la.

8.4. Atributo

Os atributos são comuns a todas as instâncias da classe e representam suas características.

```
>>>class Nome_da_classe:
>>>    atributo=quantidade_ou_qualidade_do_atributo
```

Com o comando **dir()** é possível verificar todos os atributos associados à classe.

Para acessar algum atributo do objeto basta aplicar a sintaxe:

```
>>>nome_do_objeto.nome_do_atributo
```

```
In [1]: class Alunos:
   ...:     nome = "Maria"
   ...:     turma = "302"
   ...:     serie = "terceiro ano do ensino medio"
   ...:

In [2]: Maria = Alunos

In [3]: print "Nome: ", Maria.nome
Nome:  Maria

In [4]: print "Turma de Maria ", Maria.turma
Turma de Maria  302

In [5]: print "Serie de Maria ", Maria.serie
Serie de Maria  terceiro ano do ensino medio

In [6]: dir(Maria)
Out[6]: ['__doc__', '__module__', 'nome', 'serie', 'turma']

In [7]: class Cachorro(object):
   ...:     def __init__(self,nome,idade):
   ...:         self.nome = nome
   ...:         self.idade = idade
   ...:     nome = "Bob"
   ...:     idade = 15
   ...:

In [8]: pet = Cachorro

In [9]: pet.nome
Out[9]: 'Bob'

In [10]: pet.idade
Out[10]: 15

In [11]: pet = Cachorro("Lulu", 13)

In [12]: pet.nome
Out[12]: 'Lulu'

In [13]: pet.idade
Out[13]: 13
```

8.5. Métodos

Métodos são funções definidas dentro das classes, usados para definir ações a serem executadas por uma instância desta classe. Para criar um método é necessário utilizar a sintaxe:

>>>**def** nome_do_método():

>>> <comandos>

Em Python, para tornar atributos e métodos privados basta iniciá-los por no máximo dois sublinhados e terminar com um sublinhado, as outras formas são públicas.

```
In [15]: class Triangulo(object):
    ...:     def __init__(self,base,altura):
    ...:         self.base = base
    ...:         self.altura = altura
    ...:     def area(self):
    ...:         return (self.base*self.altura)
    ...:

In [16]: t1 = Triangulo(3,2)

In [17]: print "Area ", t1.area()
Area  6
```

8.6.Herança

A herança é aplicada para aproveitar uma parte do código. As classes em Python podem herdar atributos, métodos, etc, de outras classes. Sem a utilização de herança cada um dos atributos, médotos,etc, teria que ser definido classe a classe. A sintaxe fica assim:

>>>**class** Nome_da_classe_pai:

>>> <atributos 1>

>>>**class** Nome_da_classe_filha(Nome_da_classe_pai):

>>> <atributos 2>

Neste caso a classe filha possuirá os atributos 1 e 2, os atributos 1 pois herdou da classe pai.

```
In [18]: class pai:
    ...:     Nome = "Joao"
    ...:     Residencia = "Rua das Rosas"
    ...:

In [19]: class filha(pai):
    ...:     Nome = "Alice"
    ...:     Olhos = "Verdes"
    ...:

In [20]: dir(pai)
Out[20]: ['Nome', 'Residencia', '__doc__', '__module__']

In [21]: dir(filha)
Out[21]: ['Nome', 'Olhos', 'Residencia', '__doc__', '__module__']
```

8.7.Sobreposição de Métodos

Sobreposição de métodos consiste em definir uma função na classe pai e definir outra função com o mesmo nome na classe filha, mas que possuem diferentes funcionalidades.

```
In [49]: class Car(object):
    ...:     condition = "new"
    ...:     def drive_car(self):
    ...:         self.condition = "used"
    ...:         return self.condition
    ...:

In [50]: class ElectricCar(Car):
    ...:     def drive_car(self):
    ...:         self.condition = "like new"
    ...:         return self.condition
    ...:

In [51]: carro = Car()

In [52]: carro.condition
Out[52]: 'new'

In [53]: carro.drive_car()
Out[53]: 'used'

In [54]: carro = ElectricCar()

In [56]: carro.condition
Out[56]: 'new'

In [57]: carro.drive_car()
Out[57]: 'like new'
```

9. Arquivos

Os arquivos em Python são representados pelos objetos do tipo file, que fornacem algumas formas de acesso ao arquivo, como leitura e escrita.

9.1. Open/Close

A manipulação de arquivos é realizada através do comando **open()**, onde são definidos o nome do arquivo e seu estado, se será criado ou lido. Quando o arquivo é criado, as informações nele inseridas são armazenadas no buffer e para que possam ser executadas é preciso dar o comando **close()**.

>>>variável=**open**(“nome_do_arquivo.extensão”,”modo_do_arquivo”)

>>> <comandos>

>>> variável.**close()**

9.2. Modos

Os modos de manipulação no Python são os seguintes:

- Write (w): Escreve no arquivo, caso o arquivo não exista, ele é criado no momento. O modo write sobrescreve o que o arquivo contém, apagando as informações anteriormente escritas.
- Append (a): Para que o conteúdo não seja apagado, utiliza-se o modo append, que insere novas informações no arquivo sem apagar o que nele estava anteriormente. O texto adicionado será inserido ao fim do arquivo. No caso do arquivo não existir, ele será criado.
- Read(r): O read é usado para ler o conteúdo do arquivo existente.

Caso não seja empregado algum modo no arquivo, por default é empregado o modo read.

9.3. Modificadores de modos

Os modificadores de modos são usados juntamente com os modos descritos anteriormente, possibilitando ampliar as funcionalidades da manipulação de arquivos. Os modificadores de modos são os seguintes:

- +: Permite tanto leitura quanto escrita. Por exemplo: quando usado da seguinte forma “r+”, possibilita ler e escrever no arquivo.
- b: Manipula o arquivo em binário e não em texto como é usualmente feito. Por exemplo: “rb”.

9.4. Funções aplicadas a arquivos

.WRITE(): Utilizado para escrever algo no arquivo.

>>> variável.**write**(“ Texto a ser inserido no arquivo”)

.READ(): Utilizado para ler o arquivo.

>>> variável.**read()**

.READLINE(): Utilizado para ler apenas uma linha do arquivo.

>>> variável.**readline()**

.CLOSED(): Utilizado para verificar se o arquivo foi realmente fechado. Seu retorno será um booleano, True no caso do arquivo estar fechado e False para caso esteja aberto.

>>> variável.**closed()**

9.5. With, as

Em Python é possível fazer com que o arquivo criado seja automaticamente fechado logo após os comandos sem o uso do método .close(). Para isso, utiliza-se para abrir o arquivo o comando with... as...

>>> **with open**(“nome_do_arquivo.extensão”,”modo do arquivo”) **as** variável:

>>> variável.**write**(“Texto a ser inserido no arquivo”)

10. Comandos úteis do Python

10.1. Try / Except

Quando um código foi escrito sem erros, ele funcionará corretamente, porém quando isso não ocorre, é adequado o uso de uma mensagem de exceção. Esta tarefa de inserir uma mensagem de exceção é dedicada ao programador através de try/except. Essa operação é importante pois quando ocorre uma falha é gerada uma exceção, e se esta exceção não for tratada o programa para de ser executado.

>>>try:

>>> <comandos>

>>>except:

>>> <comandos>

```
In [11]: print 1/0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-11-e19d6e6ac7e1> in <module>()
----> 1 print 1/0

ZeroDivisionError: integer division or modulo by zero

In [12]: try:
    ...:     print 1/0
    ...: except:
    ...:     print "Nao existe divisao por 0"
    ...:
Nao existe divisao por 0
```

10.2. Print com o uso da “,”

A vírgula (,) pode ser utilizada em conjunto com a função print para exibir a informação sem pular linha.

```
In [1]: a = [1,2,3,4]

In [2]: for i in range(len(a)):
   ...:     print a[i]
   ...:
1
2
3
4

In [3]: for i in range(len(a)):
   ...:     print a[i],
   ...:
1 2 3 4
```

10.3. Compreensão de listas

Supondo que se queira construir uma lista com 20 elementos. Para isso pode ser usado o comando range(20). Porém, pode-se desejar uma determinada lista com restrições e filtros. Por exemplo, os números pares de 0 até 50 ou os números divisíveis por 7 da sequência [3,158], ou até mesmo de uma maneira simplificada ter uma lista da sequência [0,50] que sejam pares.

```
In [4]: pares_0_50 = [i for i in range(51) if i % 2 == 0]
```

Poderia ser feita de outra maneira, porém não tão prática.

```
In [7]: for i in range(51):
   ...:     if i %2 == 0:
   ...:         pares_0_50.append(i)
```

A ideia geral da compreensão de listas é criar uma lista com filtros e restrições de maneira rápida e simples.

10.4. Passo e fatiamento de listas

As listas podem ser fatiadas utilizando a seguinte sintaxe:

Nome_da_lista[inicio:fim:passo]

Início seria onde o fatiamento começa. Fim onde termina, porém, não incluindo esse número. O passo é o espaço entre cada elemento do fatiamento.

```
In [11]: l = [1,4,9,16,25,36,49,64,81,100]

In [12]: l[2:9:2]
Out[12]: [9, 25, 49, 81]
```

Os índices para o fatiamento podem ser omitidos, sendo assim, o índice para o início passa a ser o padrão, no caso o índice 0, o fim da lista assume um valor determinado pelo usuário, e o passo será 1.

```
In [13]: l = [1,4,9,16,25,36,49,64,81,100]

In [14]: l[:3]
Out[14]: [1, 4, 9]
```

OBS: O passo também pode ser negativo. Sendo assim, a lista será percorrida da direita para a esquerda.

10.5. Funções Lambda

Funções Lambda ou Anônimas são funções cujo propósito é deixar o programa mais simples e funcional.

```
In [1]: f = lambda x: x*2

In [2]: f(3)
Out[2]: 6
```

Que é o mesmo que:

```
In [3]: def f(x):
   ...:     return x*2
   ...:

In [4]: f(2)
Out[4]: 4
```

As funções anônimas são muito úteis quando combinadas com a função filter(), para filtrar uma lista.

```
In [5]: lista = range(16)

In [6]: print filter(lambda x: x%3 == 0, lista)
[0, 3, 6, 9, 12, 15]
```

11. Bibliotecas

Assim como qualquer outra linguagem de programação, o Python possui bibliotecas de códigos, com funções já definidas, entre outras ferramentas.Para utilizar essas bibliotecas basta usar o comando **import** nome_do_módulo, ou o comando **from** nome_da_biblioteca **import** nome_do_módulo. Além disso, para facilitar o chamado de uma biblioteca, podemos usá-la com o nome que quisermos através do comando **as** ficando então **import** nome_do_módulo **as** nome_de_uso ou **from** nome_da_biblioteca **import** nome_do_módulo **as** nome_de_uso. Um outro comando muito útil quando se trata de bibliotecas é o dir(nome_da_bibilioteca), por mostrar todas as funções contidas nelas. Para utilizar os métodos de uma biblioteca basta chamá-la como nome_da_biblioteca.nome_da_função(argumentos).

```
In [1]: import math

In [2]: import math as mt

In [3]: from math import sqrt

In [4]: sqrt(4)
Out[4]: 2.0

In [5]: mt.sqrt(4)
Out[5]: 2.0
```

Lembrando que quando é utilizado o comando **import,** este buscará o módulo dentro da biblioteca, caso não seja encontrado, ele passará a interpretar o que estiver após o **import** como sendo uma biblioteca. Caso a biblioteca não exista ocorrerá um erro.

Serão mostradas as principais utilizações de algumas bibliotecas do Python, algumas já instaladas, outras não.

11.1. Math

A biblioteca math tem como objetivo facilitar o uso da linguagem matemática. Funções como raiz quadrada, valor de pi e etc já estão embutidas nela.

.FACTORIAL(x): Retorna o fatorial de x.

```
In [2]: math.factorial(5)
Out[2]: 120
```

.EXP(x): Retorna o valor de e^x.

```
In [3]: math.exp(2)
Out[3]: 7.38905609893065
```

.SQRT(x): Retorna o valor da raix quadrada de x.

```
In [4]: math.sqrt(8)
Out[4]: 2.8284271247461903
```

.CEIL(x): Retorna o menor valor inteiro maior que x.

```
In [8]: math.ceil(5.3)
Out[8]: 6.0
```

.FLOOR(x): Retorna o maior valor inteiro menor que x.

```
In [9]: math.floor(5.3)
Out[9]: 5.0
```

.LOG(x,[b]): Retorna valor do log de x na base b. Caso não seja dado o valor de b, considera-se b = e(número de Euller).

```
In [10]: math.log(5,3)
Out[10]: 1.4649735207179269

In [11]: math.log(5)
Out[11]: 1.6094379124341003
```

.PI / .E: Retorna os valores das constante pi e “e” (número de Euller).

```
In [12]: math.pi
Out[12]: 3.141592653589793

In [13]: math.e
Out[13]: 2.718281828459045
```

.DEGREES(x): Retorna o valor do ângulo x de radiano para grau.

```
In [14]: math.degrees(math.pi)
Out[14]: 180.0
```

.RADIANS(x): Retorna o valor do ângulo x de grau para radiano.

```
In [15]: math.radians(180)
Out[15]: 3.141592653589793
```

.SIN(x) / .COS(x) / .TAN(X): Retorna o seno/cosseno/tangente de x.

```
In [16]: math.sin(math.pi/4)
Out[16]: 0.7071067811865475
```

.ASIN(x)/.ACOS(x)/.ATAN(x): Retorna o valor de arco seno/ cosseno/ tangente de x.

```
In [17]: math.degrees(math.asin(math.sin(math.pi/4)))
Out[17]: 44.99999999999999
```

11.2. Random

O módulo Random tem como proposta pegar algum elemento aleatório de algum tipo de dado.

.RANDOM(): Retorna um elemento aleatório entre 0 e 1.

```
In [2]: random.random()
Out[2]: 0.253574836837945
```

.UNIFORM(a,b): Retorna um elemento aleatório entre a e b, incluindo a e excluindo b.

```
In [7]: random.uniform(1,5)
Out[7]: 2.667357697671913
```

.RANDINT(a,b): Retorna um elemento inteiro aleatório entre a e b incluindo esses dois.

```
In [3]: random.randint(1,4)
Out[3]: 4
```

.RANDRANGE(b)/(a,b,passo): Retorna um elemento inteiro aleatório entre a e b, incluindo a e excluindo b, com um passo determinado.

```
In [8]: random.randrange(1,5,2)
Out[8]: 3
```

.CHOICE(seq): Retorna um elemento aleatório de uma sequência seq.

```
In [5]: random.choice([1,2,3,4,6,8])
Out[5]: 1
```

.SAMPLE(seq, k): Retorna uma lista de tamanho k com k elementos aleatórios de uma sequencia seq.

```
In [6]: random.sample([12,36,58],2)
Out[6]: [36, 58]
```

11.3. Datetime

A biblioteca Datetime permite trabalhar com dia e tempo.

DATETIME.NOW(): Coleta o dia e tempo correntes.

>>> print datetime.now()

>>> 2015-11-08 01:58:02.182000

DATETIME.NOW().YEAR: Destaca da data apenas o ano.

>>> print datetime.now().year
>>> 2015

DATETIME.NOW().MONTH: Destaca da data apenas o mês.

>>> print datetime.now().month
>>> 11

DATETIME.NOW().DAY: Destaca da data apenas o dia.

```
>>> print datetime.now().day
>>>8
```

DATETIME.NOW().HOUR: Destaca do tempo apenas a hora.

```
>>> print datetime.now().hour
>>> 1
```

DATETIME.NOW().MINUTE: Destaca do tempo apenas o minuto.

```
>>> print datetime.now().minute
>>>58
```

DATETIME.NOW().SECOND: Destaca do tempo apenas o segundo.

```
>>> print datetime.now().second
>>> 18
```

11.4. Numpy

Numpy é um pacote que tem como principal função realizar operações com vetores e matrizes. Essa ferramenta é muito superior às operações com vetores originais do Python. O tipo de variável criado pela biblioteca Numpy é chamado de “array”

.ARRAY([[Linha1],[Linha2],...,[LinhaN]]): método principal de criação de matrizes e vetores. Perceba que seus argumentos são listas dentro de uma única lista. Caso seja um vetor, pode-se fazer da seguinte forma: .ARRAY([linha1]).

```
In [4]: a = numpy.array([1,2,3,4,5])

In [5]: print a
[1 2 3 4 5]

In [6]: a = numpy.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])

In [7]: print a
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
```

.SHAPE: Retorna o tamanho de cada dimensão da matriz.

```
In [10]: a.shape
Out[10]: (3L, 5L)
```

.TRANSPOSE(): Retorna a transposta da matriz.

```
In [15]: a.transpose()
Out[15]:
array([[ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14],
       [ 5, 10, 15]])
```

.SUM(matriz,[axis =(0,1)]): Retorna a soma de todos elementos de uma matriz caso só tenha um argumento. No caso contrário, se colocado axis = 0, retornará uma lista com a soma dos elementos de cada coluna, se for axis = 1, retornará uma lista com a soma dos elementos de cada linha.

```
In [16]: a
Out[16]:
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])

In [17]: numpy.sum(a)
Out[17]: 120

In [18]: numpy.sum(a,axis = 0)
Out[18]: array([18, 21, 24, 27, 30])

In [19]: numpy.sum(a,axis = 1)
Out[19]: array([15, 40, 65])
```

.ARANGE(c): Muito parecido com a função range. A grande diferença que retorna uma variável do tipo array e não uma lista.

```
In [20]: numpy.arange(12)
Out[20]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

.ZEROS([a,b], dtype = Int): Cria uma matriz com dimensões axb preenchida por zeros do tipo float caso só tenha dado um argumento. No segundo argumento há a especificação de inteiro (dtype = int).

```
In [21]: numpy.zeros([2,3])
Out[21]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

.LINALG.DET(a): Retorna o determinante de uma matriz a.

```
In [33]: np.linalg.det(a)
Out[33]: -2.0000000000000
```

11.4.1. Acessando elementos

Da mesma forma como as listas, pode-se acessar cada elemento de um array. De forma análoga, a maneira de se fazer isso é matriz[Indice_linha,Indice_coluna].

OBS: Não se esquecer que no Python sempre se começa a contar os índices a partir do 0. Então para acessar o elemento da linha 2, coluna 3 deve-se por array[1,2]

```
In [22]: a
Out[22]:
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])

In [23]: a[1,2]
Out[23]: 8
```

11.4.2. Operações entre arrays

Uma das grandes vantagens do uso do numpy está no fato das suas operações já estarem previamente programadas. Os comandos das operações com arrays seguem o mesmo padrão das operações aritméticas básicas na soma e na subtração, sendo a multiplicação de matrizes feita com o comando .DOT(a,b), onde a e b são matrizes.

```
In [34]: a
Out[34]:
array([[1, 2],
       [3, 4]])

In [35]: a+a
Out[35]:
array([[2, 4],
       [6, 8]])

In [36]: a*3
Out[36]:
array([[ 3,  6],
       [ 9, 12]])

In [37]: numpy.dot(a,a)
Out[37]:
array([[ 7, 10],
       [15, 22]])
```