Módulos Didáticos

COMECE A PROGRAMAR COM PYTHON

LINKED EDUCATION

V1017

Sumário

Módulos Didáticos

Módulo 01

COMECE A PROGRAMAR COM PYTHON

LINKED EDUCATION

Sobre Python

Rápido aprendizado

Menos código que outras linguagens para realizar a mesma tarefa

Ampla abrangência de utilização

Multiparadigma...e muito mais.

Algoritmos

Algoritmos

- Imagine uma "máquina abstrata" *M* capaz de realizar "operações" que manipulam "coisas".
- Informalmente, um algoritmo é uma sequência de operações de tal máquina.
- As "coisas" são tipicamente chamadas de objetos.
- Um "problema" para *M* consiste de dois ingredientes: uma entrada e uma saída.
- Tanto a entrada quanto a saída são constituídas de 0 ou mais objetos que a máquina pode manipular.
- O problema deve especificar uma *relação* entre a entrada e a saída.
- Um algoritmo para *M* é uma sequência de operações de *M* capaz de "transformar" a entrada na saída.
- Nem sempre uma tal transformação existe.

Algoritmos e Programas

Algoritmos e Programas

Conteúdo

- Um programa é um algoritmo para uma "máquina concreta".
- Programas tipicamente estão associados aos computadores modernos.
- Programas são escritos em linguagens de programação.
- Há uma ampla gama de linguagens de programação: C, C++, Java, Prolog, Haskell, Scala, Python, etc.
- As linguagens de programação fornecem uma maneira se de comunicar com as operações fornecidas pela máquina de uma forma mais amigável.
- Mais do que isso. Elas fornecem abstrações que permitem a solução de problemas de uma forma mais elegante.
- A escolha de uma linguagem de programação delimita a capacidade de expressão na hora de resolver problemas.
- IMPORTANTE: Não existe uma melhor linguagem de programação.

Modelo Abstrato de Computador

Modelo Abstrato de Computador

Você pode pensar em um computador como constituído de duas partes:

- a primeira parte é responsável pela implementação das operações realizadas pela máquina;
- a segunda, tipicamente chamada de memória, pode ser pensada como uma lista $\langle m_1, m_2, \ldots, m_n \rangle$ capaz de armazenar os tais objetos que os programas manipulam.
- Cada m_i é chamado de célula.

Em particular, os próprios programas são armazenados nesta mesma memória.

Modelo Abstrato de Computador

- Cada m_i é uma sequência de 0's e 1's de tamanho fixo.
- Uma linguagem de programação nos permite ver a memória de uma forma mais abstrata, onde cada m_i pode alternativamente "armazenar" os objetos fornecidos direta ou indiretamente pela linguagem.
- Objetos em uma linguagem de programação possuem um atributo chamado de tipo.
- Um tipo nada mais é que uma certa sequência de 0's e 1's dotada de um certo padrão.
- Estes padrões são escondidos dos usuários das linguagens.
- Esta noção é violentamente importante!

Iniciando com Python

- Uma *computação* em PYTHON envolve objetos.
- Computações são realizadas em objetos e produzem outros objetos como resultado.
- Por exemplo, se você digitar a expressão “3+5” no interpretador do Python, ele avaliará esta expressão e produzirá o objeto 8.
- Todo objeto em Python possui um tipo.
- Um tipo, além de um nome que o identifica, possui dois atributos importantes:
  1. um conjunto de objetos possíveis, e
  2. um conjunto de operações sobre estes objetos.

Inteiros

O primeiro objeto que vamos estudar são os inteiros que, mais precisamente, deveríamos chamar de int's.

Não é de surpreender que a forma como um int é representado seja similar a representação que você aprendeu na escola.

```
>>> 28
28
```

Após a digitação de 28, em uma certa célula de memória será armazenado o número 28. Se você quiser saber o tipo de um objeto, use a função type:

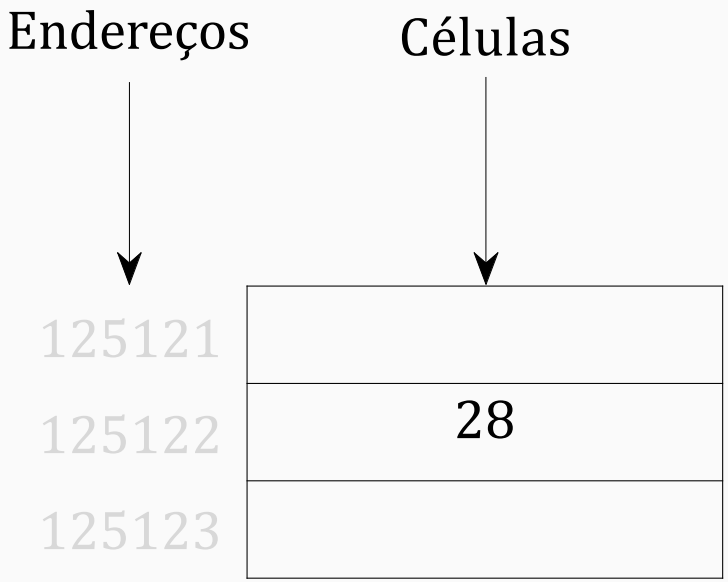
```
>>> type(28)
<class 'int'>
```

Isto quer dizer que 28 é uma instância (exemplo) do tipo int. Note que a sequência de dígitos 28 é a representação do número 28 em Python. Esta sequência é um exemplo de uma constante.

```
> > > id(28)
125122
```

A função id devolve o identificador do objeto fornecido como argumento.

Tal identificador pode ser o endereço de memória do objeto.



Algumas Operações sobre **int´s**

- A adição, subtração, multiplicação, e divisão (inteira) são respectivamente representadas pelos operadores “+”, “−”, “∗” e “//”.
- As operações acima são ditas binárias, pois envolvem dois argumentos.
- Em particular estas operações recebem um par ordenado de int’s e produzem um int, escrito int × int → int.
- O valor produzido por uma operação é, como já mencionado, armazenado em uma das células da memória.

- A sequência de caracteres $3 + 5 * 5$ é uma *expressão*.
- Em geral a combinação de operandos e operadores produzem o que chamamos de uma expressão.
- Assim, expressões "maiores" podem ser obtidas de expressões "menores", chamadas de subexpressões, usando operadores.
- Avaliar uma expressão significa determinar o objeto ao qual a expressão se refere.
- Por exemplo, a expressão $3 + 4 * 5$ é avaliada da seguinte forma:

$$3 + 4 * 5 \mapsto 3 + 20 \mapsto 23.$$

- Aqui, $x \mapsto y$ é uma forma de dizer que a expressão x é avaliada e a expressão y é obtida.

- Uma expressão como $3 + 4 * 5$ pode, em princípio, ser entendida de duas formas diferentes: $3 + (4 * 5)$ ou $(3 + 4) * 5$.
- Em Python, a escolha de interpretação é a mesma usada habitualmente em Matemática.
- Assim, há uma precedência envolvendo os operadores.
- Como é de se imaginar, a interpretação escolhida é $3 + (4 * 5)$

- **Nomes** (ou identificadores) possuem um papel central na programação.
- Podemos associar um nome a um objeto.
- Isto é realizado usando-se o operador de atribuição, =.
- Uma regra estabelece quais sequências de caracteres são nomes.
- Assim, uma sequência de caracteres representa um nome se começa com uma letra (maiúscula ou minúscula) ou um _ e é seguido de uma sequência de 0 ou mais letras, dígitos ou _

- São exemplos de nomes: *x*, *soma*, *total*.

- Uma atribuição tem a forma $x = e$, onde x é um nome, e e é uma expressão.
- Primeiro a expressão e é avaliada, obtendo-se um certo objeto v, que reside em alguma célula da memória.
- Em seguida, o nome x é associado (ou ligado) ao objeto v.
- Por exemplo,

```
>>> x = 5
>>> x
5
>>> id(5)
 10437344
>>> id(x)
10437344
```

Eis uma representação:

Endereço	Células
10437343	
10437344	5
10437345	

x

Considere o seguinte exemplo

```
>>> x = 5
>>> x
5
>>> id(5)
10437344
>>> id(x)
 10437344
>>> x = 10
>>>
10
>>> id(10)
 10437346
>>> id(x)
 10437346
```

Eis uma representação:

Endereço	Células	
10437343		
10437344	5	x
10437345		
10437346	10	x
10437347		

Primeiros programas com int´s

Faremos um programa que soma a um inteiro (no exemplo é 3, associado ao nome m) o dobro dele.

```
m=3
n=3+2*m
print(n)
```

Resultado :

```
>>> %Run teste.py
9
>>>
```

Primeiros programas com int´s

Faça um programa com dois int´s (associados a m e n). Elas receberão por atribuição respectivamente dois números inteiros. O programa deve calcular a soma delas e atribuí-la a um terceiro int (associado a k). Finalmente, o programa apresentará o resultado na tela.

O operador **//** faz a divisão inteira entre dois int´s. Já o operador **%** traz o resto da divisão entre dois int´s **.** Por exemplo :

9	***2***
1	***4***

Faremos um programa que recebe dois números inteiros (associados a m e n) e apresenta o valor da divisão inteira de m por n e o resto dessa divisão.

```
m=9
n=4
print(m//n)
print(m % n)  # teste-o (isto é um comentário)
```

Float´s (ponto flutuante)

O Python fornece três objetos imutáveis para representar números de ponto flutuante cujos tipos são:

- float
- complex
- decimal.Decimal

Os float's são números de ponto flutuante com dupla-precisão (normalmente até 17 dígitos significativos).
O número de bits da representação é fixo.
Possuem uma precisão limitada.
O operador de igualdade não é confiável.
Exemplos: 5.9, 5e10, 4e-2
As operações +, -, *, / operam como esperado.

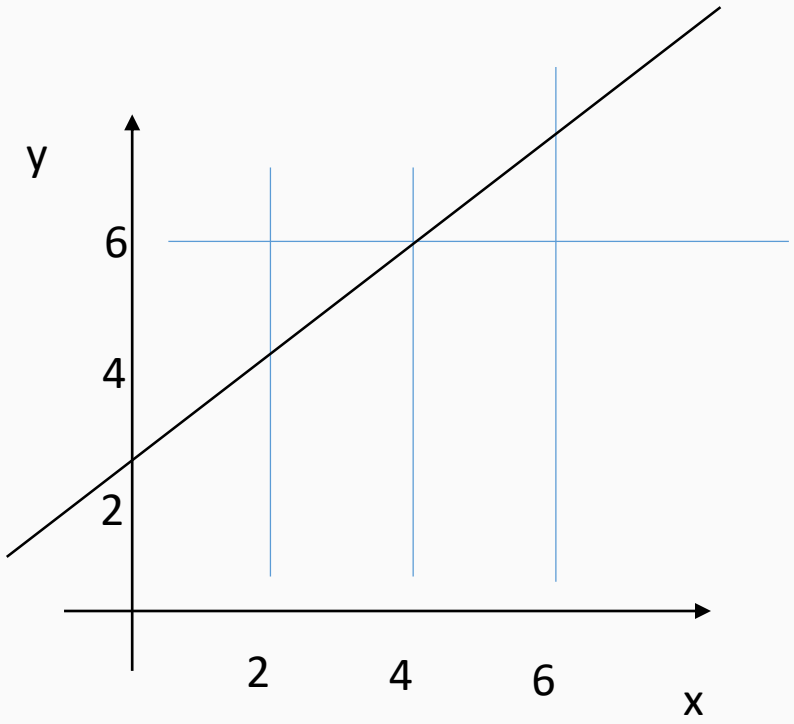
Se for necessária uma precisão maior, utilizamos decimal.Decimal.
Para objetos deste tipo é possível fixar a precisão.
O processamento é mais lento que o envolvendo float's.

```
>>> import decimal
>>> a = decimal.Decimal('42.563638290282879383837')
>>> b = decimal.Decimal('56.76655453242342343243434')
>>> a+b
Decimal('99.330192822706302816271 34')
```

Primeiros exemplos com floats

A equação da reta ao lado é :
y= 0.875*x+2.5

Faremos um programa em Python para saber o valor de y para x=6




```
x=6
y=0.875*x+2.5
print(y)
```

Teste-o

Exercício

Na matemática financeira, um valor **v_inicial** corrigido à taxa mensal de juros (compostos) i (%) tem seu valor, após **n** meses :

v_corrigido= **v_inicial***(1 + i*0.01)**n

Faça um programa Python que recebe o valor_inicial, a taxa de juros e o número de meses. O programa deve apresentar o valor corrigido

Strings

Sequências de caracteres ou strings são representadas por objetos imutáveis do tipo str.
Literais do tipo str são criadas envolvendo-se a sequência de caracteres em aspas simples ou duplas.

```
>>> 'Ola, mundo!'
'Ola, mundo!'

>>> "Ola, mundo!"
'Ola, mundo!'
```

Strings podem ser concatenadas usando-se o operador +:

```
>>> s1 = 'Ola, '
>>> s2 = 'mundo!'
>>> s1 + s2
'Ola, mundo!'

>>> s = s1 + s2
>>> s
'Ola, mundo!'
```

O comprimento de uma string pode ser obtida usando-se a função len:

```
>>> len(s)
11
```

Podemos, então, escrever s[expr], onde s é um objeto do tipo str e expr é uma expressão
aritmética desde que expr, quando avaliada, tenha um dos valores 0, 1, ..., len(s)-1.
O valor produzido por s[expr] é uma string de comprimento 1 que consiste do elemento
s[i], onde i é o valor de expr.

O conjunto dos valores 0, 1, ..., len(s)-1 é chamado de conjunto dos índices de s.

Observação: é fundamental desde já se acostumar a contar a partir do 0!

A operação de selecionar uma determinada parte de uma string é conhecida como fatiamento.

Há uma gama enorme de operações de fatiamento em Python.

Vamos destacar algumas: (s="olá, mundo!")

```
>>> s[2:7]
'a, mu'

>>> s[1:8]
'la, mun'
```

Observação: Intervalos em Python são fechados na esquerda e abertos na direita.

Assim, s[2:7] é o fatiamento produzido concatenando-se s[2], s[3], s[4], s[5], s[6].

É muito importante se acostumar com a ideia de intervalos fechados na esquerda e abertos na direita.

Você pode até desconfiar, mas esta é a forma correta!

Lembre-se que o Python é filho de um matemático, enquanto que o Java, é de engenheiros... :-)

Primeiros exemplos com strings

Criaremos a string associada ao nome s1=“Brasil é o país do futuro”

Criaremos a string s2 retirando (por fatiamento) a string “país” de s1

Apresentaremos s2 na saída padrão (com print())

```
s1="Brasil é o país do futuro"
s2=s1[11:15]
print(s2)

#teste-o!
```

Desafio

Crie uma string associada a s1=“João é bom menino”

Crie uma string s2=“Carlos tem medo de bola”

Crie uma string s3 concatenando uma fatia de s1 e uma fatia de s2 para gerar “João é bom de bola”

Apresente s3 (print(s3))

Uma **lista** é uma sequência de 0 ou mais referências para objetos.

A lista vazia é escrita [] e seu comprimento é 0:

```
>>> len([])
0
```

Note a beleza. Você utiliza a 'mesma' função len para determinar o comprimento de uma lista.

Tais uniformidades constituem uma das características marcantes do Python.

Uma lista mais 'complicada':

```
>>> len([1, 4, 5, 2])
4
```

Em geral escrever [e1, e2, ..., ek] produz uma lista com k elementos [v1, v2, ..., vk],
onde e1, e2, ..., ek são expressões tais que ei |-> vi para cada i:

```
>>> xs = [1+4, 5, 6*7, 8*7]
>>> xs
[5, 5, 42, 56]
```

É possível construir uma lista com objetos de quaisquer natureza, inclusive listas:

```
>>> vs = ['Ola', 2, 5.7, 'Mundo']
>>> vs
['Ola', 2, 5.7, 'Mundo']
```

Listas, diferentemente dos outros objetos que já vimos, são mutáveis. Podemos modificar o seu conteúdo, inserir e remover elementos.

```
>>> xs
[5, 5, 42, 56]
>>> xs[1] = 10
>>> xs
[5, 10, 42, 56]
```

É possível inserir um novo elemento no final da lista usando append:

```
>>> xs.append(20)
>>> xs
[5, 10, 42, 56, 20]
```

Podemos também remover um elemento do final da lista usando pop:

```
>>> xs.pop()
20
>>> xs
[5, 10, 42, 56]
```

Nas listas também podemos fazer fatiamento

```
>>> lista=[9,"ovo",4,6,7]
>>> lista[1:4]
['ovo', 4, 6]
```

E também podemos alterar os elementos dela

```
>>> lista=[9,"ovo",4,6,7]
>>> lista[1:4]
['ovo', 4, 6]
>>>
>>> lista[1]="galinha"
>>> lista
[9, 'galinha', 4, 6, 7]
```

Desafio : Explique !

```
>>> a=10
>>> id(a)
1687246720
>>> a=11
>>> id(a)
1687246736
>>> l=[1,4,5]
>>> id(l)
44070832
>>> l[0]=2
>>> l
[2, 4, 5]
>>> id(l)
44070832
```

Primeiros exemplos com listas

Vamos criar uma lista com a string “leão”, depois colocaremos mais 3 animais e apresentaremos a lista. Depois retiraremos o último animal colocado na lista e a reapresentaremos.

```
>>> animais=["leão"]
>>> animais.append("Girafa")
>>> animais.append("Zebra")
>>> animais.append("Elefante")
>>> animais
['leão', 'Girafa', 'Zebra', 'Elefante']
>>> animais.pop()
'Elefante'
>>> animais.pop()
'Zebra'
>>> animais.pop()
'Girafa'
>>> animais
['leão']
```

Exercício

Faça uma lista de números e carros. Depois, construa, a partir da primeira uma lista só de carros

Iterações (laços)

As sentenças de um programa são executadas
uma por vez, de cima para baixo.

É possível controlar o fluxo de execução.

Um dos principais padrões de controle de fluxo de execução
é a iteração (ou laço) que permite executar uma
mesma seção de código diversas vezes.

O Python possui uma riquíssima coleção de iteradores,
que permitem a escrita de códigos extremamente elegantes.

Iterações (laços)

Vamos começar com uma versão muito básica do comando **for**.

Você vai aprender um grande número de variantes deste comando durante o curso.

Para que seja possível utilizá-lo em uma de suas formas mais básicas, vamos precisar tomar contanto com a noção de um range --- que é uma expressão geradora.

Um objeto range é violentamente útil em iterações.
para um certo int n, range(n) produz um objeto range
que gera, nesta ordem, a sequência de ints de 0 até n-1.

Primeiros exemplos de **for**

```
for i in range(5):
    print(i)
```

Resultado :

```
0
1
2
3
4
```

Primeiros exemplos de **for**

```
for i in range(2,5):
        print(i)
```

Resultado :

```
2
3
4
```

Exercício

Faça um laço for que apresente o quadrado de 1 até 10

1
4
9...

Primeiros exemplos de **for**

```
minha_lista=[2,4,'sabão',50.3]
for i in minha_lista:
    print(i)
```

Resultado :

```
2
4
sabão
50.3
```

Exercício

Faça uma lista de nomes e, com um laço **for** apresente os nomes um a um concatenados com “eu conheço”...

João eu conheço
Maria eu conheço
Paulo eu conheço

Módulo 02

COMECE A PROGRAMAR COM PYTHON

Sobre o Módulo 1

Revisão

- ✓ Computadores, Algoritmos e Programas
- ✓ Int
- ✓ Float
- ✓ String
- ✓ List

Sobre o Módulo 1

Vamos revisar e conceituar melhor os **Laços**, na sua variante **for** :
O comando **for** tem a forma:

```
for <nome> in range(<expr>):
    <stmt1>
    <stmt2>
    .............
    <stmtk>
```

Sobre o Módulo 1

Revisão

for <nome> **in range**(<expr>):
    <stmt1>
    <stmt2>
    .....
    <stmtk>

Como funciona :
1)A expressão <expr> é avaliada, gerando um int **n**

2)O objeto **range** gera uma sequência de inteiros (0 a n-1)

3)<nome> assume o primeiro valor da sequência gerada

4)As sentenças <stmt1> até <stmtk> são executadas com <nome> tendo o primeiro valor da sequência

5)O processamento continua até <nome> assumir o n-ésimo valor da sequência (n-1) e as sentenças <stmt1> a <stmtk> serem processadas com esses valores....depois disso o processamento continua no primeiro comando depois de <stmtk>

Exemplo de laço

```
for i in range(2):
    print(i)
    print(2*i)
print('acabou o laço')

>>>
0
0
1
2
acabou o laço
```

Observe : range(2) gerou 2 inteiros 0 e 1
Os comandos print(i) e print(2*i) foram executados com i assumindo os valores 0 e 1
Finalizado o laço, foi executado o commando :
print('acabou o laço')

Entrada e saída

Vamos destacar duas operações básicas de entrada e saída.

Saída

Já vimos que a função **print**(<expr>) é responsável por exibir valor de <expr> na tela.

Entrada

Como devemos fazer para ler uma informação digitada pelo usuário? Para isso, podemos usar a função **input**(str), cujo argumento é um literal string...veja o exemplo no próximo slide

Entrada e saída

Conteúdo

teste.py *

```
nome=input('entre com seu nome ')
print(nome)
```

```
>>> %Run teste.py
 entre com seu nome
```

1)Ao ser executado, o comando input('entre com seu nome ‘) apresenta a mensagem *entre com seu nome* e aguarda até o usuário digitar o nome e teclar Enter

```
>>> %Run teste.py
 entre com seu nome John Von Neumann
```

2)Depois de digitar a string e teclar Enter, o objeto string digitado recebe o nome **nome.** Finalmente, o objeto é apresentado com o comando de saída **print**(nome) **.**

```
>>> %Run teste.py
entre com seu nome John Von Neumann
John Von Neumann
```

Entrada e saída

Mais um exemplo de Entrada e Saída

teste.py

```
idade=int(input('entre com sua idade '))
print(idade)
```

```
>>> %Run teste.py
 entre com sua idade |
```

Nesse exemplo, a string digitada é convertida para o tipo **Int** e só depois é associada ao nome **idade**. Finalmente, o objeto é apresentado com o comando **print**(idade)

```
>>> %Run teste.py
 entre com sua idade 18
 18
```

Combinando entrada com **input** e laço **for**

```
quant=int(input('entre com o número de itens '))
soma=0
for i in range(quant):
    item=int(input('entre com o valor do item '))
    soma=soma+item # ou também soma+=item
print(soma)
```

```
Shell
>>>
>>> %Run teste.py
 entre com o número de itens 3
 entre com o valor do item 3
 entre com o valor do item 7
 entre com o valor do item 9
 19
```

Nesse exemplo, é lida a quantidade de itens a serem lidos e, a cada iteração do laço, o valor de um item é lido e acumulado em soma. Finalmente, o valor da soma é apresentado.

O tipo booleano

Há dois objetos no tipo bool: **True** e **False**.

Como são objetos, podemos associar-lhes nomes. Logo, é válido escrever:

x = True

ou

x = False

O Python oferece três operadores lógicos:

and,

or

e

not.

Os operadores and e or são binários, ou seja, operam sobre dois objetos do tipo bool (ou que podem ser convertidos para o tipo bool).O operador not é unário, ou seja, opera sobre um único objeto do tipo bool

O tipo booleano

Eis como estes operadores **and, or** e **not** se comportam:

<expb1> **and** <expb2> é True se e somente se ambos <expb1> e <expb2> quando avaliadas são True.

<expb1> **or** <expb2> é False se e somente se ambos<expb1> e <expb2> quando avaliadas são False.

not <expb> é True se <expb> quando avaliada é False.
not <expb> é False se <expb> quando avaliada é True.

Operadores relacionais

Há um conjunto de operadores binários, chamados de **relacionais**,que produzem objetos do tipo **bool**.

Eis alguns exemplos:<exp1> **==** <exp2> é True se e somente se <exp1> e <exp2> quando avaliadas possuem o mesmo **valor**.

<exp1> **!=** <exp2> é o mesmo que **not** (<exp1> == <exp2>).

<exp1> **>=** <exp2> é True se e somente se o valor de <exp1> quando avaliada é maior ou igual ao valor de <exp2> quando avaliada.

Analogamente temos os operadores **>**, **<** e **<=** **.**

Booleanos e relacionais

Exercício

Um banco só fornece crédito para quem tem mais de 30 anos e salário acima de R$2000.

Faça um programa em Python que tem como entrada a idade e o salário do cliente e retorna True ou False caso o cliente tenha ou não crédito, respectivamente

Booleanos e relacionais

Exercício

Resolução (e teste para duas condições diferentes de entrada)

teste.py

```
idade=int(input('Entre com sua idade '))
salario=float(input('Entre com seu salário '))
print((idade > 30) and (salario > 2000))
```

Shell

```
>>>
>>>
>>> %Run teste.py
 Entre com sua idade 35
 Entre com seu salário 1990
 False
>>> %Run teste.py
 Entre com sua idade 31
 Entre com seu salário 2500
 True
>>>
```

Comando de decisão **IF**

Uma outra forma (além dos laços) de alterar o fluxo de execução é por meio dos comandos de decisão.

O comando if tem a seguinte forma:

```
if   <cond>:
  <cmd1>
  <cmd2>
.............
  <cmdk>
```

A semântica do comando acima é a seguinte.

1. A condição é avaliada.
2. 2. Se <cond> avaliada é True, então os comandos <cmd1>, <cmd2>, ..., <cmdk> são executados nesta ordem; após isso, a execução continua a partir do próximo comando que segue o comando if.
3. 3. Se <cond>avaliada é False, então a execução continua a partir do próximo comando que segue o comando if.

Comando de decisão **IF**

Exemplo : Faça um programa que lê dois Int´s e apresenta o maior deles

```
x = int(input('Entre com o primeiro número: '))
y = int(input('Entre com o segundo número: '))
if x >= y:
    print('Um maior é ', x)
if x < y:
    print('Um maior é ', y)
```

Shell

```
>>>
>>>
>>>
>>> %Run teste.py
 Entre com o primeiro número: 20
 Entre com o segundo número: 12
 Um maior é  20
```

Comando de decisão **IF-ELSE**

O comando if-else tem a forma:

If <cond>**:**

    <cmdi1>
    <cmdi2>
    .....
    <cmdik>

else:

    <cmde1>
    <cmde2>
    .....
    <cmdel>

A semântica do comando acima é a seguinte:

1. A condição <cond> é avaliada
2. 2. Se <cond> avaliada é True, então os comandos <cmd1>, <cmd2>, ..., <cmdk> são executados nesta ordem; após isso, a execução continua a partir do próximo comando que segue o comando if-else.
3. 3. Se <cond> avaliada é False, então os comandos <cmde1>, <cmde2>, ..., <cmdel> são executados nesta ordem; após isso, a execução continua a partir do próximo comando que segue o comando if-else.

Comando de decisão **IF-ELSE**

O comando if-else tem a forma:

If <cond>**:**

    <cmdi1>
    <cmdi2>
    .....
    <cmdik>

else:

    <cmde1>
    <cmde2>
    .....
    <cmdel>

A semântica do comando acima é a seguinte:

1. A condição <cond> é avaliada
2. 2. Se <cond> avaliada é True, então os comandos <cmd1>, <cmd2>, ..., <cmdk> são executados nesta ordem; após isso, a execução continua a partir do próximo comando que segue o comando if-else.
3. 3. Se <cond> avaliada é False, então os comandos <cmde1>, <cmde2>, ..., <cmdel> são executados nesta ordem; após isso, a execução continua a partir do próximo comando que segue o comando if-else.

Comando de decisão **IF-ELSE**

Exemplo : Faça um programa que lê dois Int´s e apresenta o maior deles (versão com IF-ELSE, mais “natural”)

teste.py

```
x = int(input('Entre com o primeiro número: '))
y = int(input('Entre com o segundo número: '))
if x >= y: print('Um maior é ', x)
else: print('Um maior é ', y)
```

Shell

```
>>>
>>> %Run teste.py
 Entre com o primeiro número: 10
 Entre com o segundo número: 1
 Um maior é  10
```

Comando de decisão **IF-ELIF**

O comando IF-ELIF tem a forma :

if <cond1>:
  <cmd11>
  <cmd12>
.....
  <cmdi1k1>
elif <cond2>:
  <cmde21>
  <cmde22>
.....
  <cmde2k2>
elif <cond3>:
 <cmde31>
 <cmde32>
.....
 <cmde3k3>...
else: #else é opcional
  <cmde1>
  <cmde2>
  <cmden>

A semântica do comando acima é a seguinte.

Os comandos executados são os que correspondem à primeira avaliação True de alguma condição <condi>

Pro exemplo, se a primeira <condi> avaliada True for <cond2>, os comando executados serão

 <cmde21>
 <cmde22>
.....
 <cmde2k2>

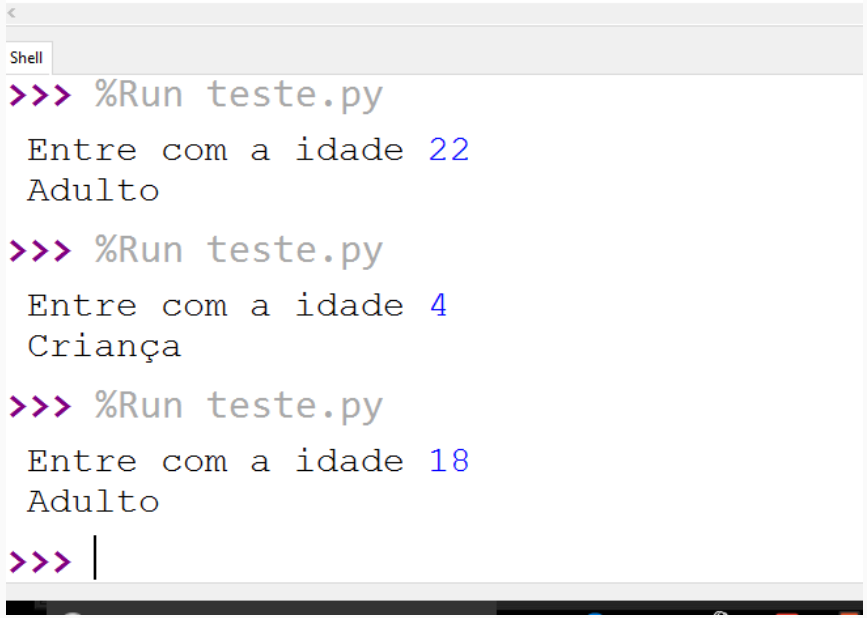
Depois o processamento prossegue para o primeiro comando depois do IF-ELIF

Se nenhuma condição <condi> avaliada for True, são executados os comandos associados ao **else**, que é opcional.

Comando de decisão **IF-ELIF**

Escreva um programa que lê a idade e a avalia da seguinte forma :
idade menor de doze anos : criança
Idade acima de doze e menor de dezoito : adolescente
Idade maior ou igual a dezoito : adulto

```
idade=int(input('Entre com a idade '))
if(idade < 12):print('Criança')
elif(idade < 18):print('Adolescente')
else:print('Adulto')
```



Laço **WHILE**

Para repetir um conjunto de comandos, além do **for**, podemos usar também o **while.** O comando **while** tem a forma:

while <cond> :
  <cmd1>
  <cmd2>
.....
  <cmdk>

A semântica do comando é : se a condição <cond> for avaliada True, os comandos <cmd1>, <cmd2> até <cmdk> são executados nessa ordem. Em seguida, a condição <cond> é novamente avaliada e se for True, os comandos serão novamente executados...essa repetição prossegue até <cond> ser avaliada como False depois de executados os comandos <cmd1> até<cmdk>

Laço **WHILE**

Exemplo : faça um programa que apresenta os números pares menores que 8

```
i=0
while (i<8):
    print(i)
    i+=2
```

Shell

```
>>>
>>>
>>>
>>>
>>>
>>> %Run teste.py
 0
 2
 4
 6
```

Módulo 03

COMECE A PROGRAMAR COM PYTHON

Sobre o Módulo 2

Revisão

Tipo booleano

IF, IF-ELSE, IF-ELIF

Laços While

Funções

Uma função em matemática é uma forma de associar elementos de dois conjuntos.

Escrevemos f: A->B para indicar que uma função leva elementos do conjunto A no conjunto B. Por exemplo, a função, que vamos chamar de f, e que leva cada inteiro x, chamado de argumento da função, em seu quadrado, x^2, é tipicamente escrita da seguinte forma: f(x) = x^2.Assim,

f(2) = 4 e

f(8) = 64.

Podemos escrever expressões mais complicadas envolvendo funções. Por exemplo, f(2) + f(4)que produz o valor 20.Uma função pode ter mais de um argumento. Por exemplo, podemos definir soma(x, y) = x + y como a função que leva inteiros x e y em sua soma, x+y.Tal função possui dois argumentos.

Funções

As linguagens de programação fornecem uma construção que permite definir restrição da noção de função em matemática. Informalmente, uma função pode ser pensada como uma sequência de comandos nomeada à qual quando fornecidos argumentos como entrada produz uma certa saída.

Uma função é um objeto que pode ser chamado. Isto quer dizer que quando escrevemos seu nome e lhe fornecemos os argumentos que ela necessita como entrada, esta é executada até o seu final produzindo um determinado valor. A partir daí novos comandos podem ser executados.

Definindo funções em Python : Eis uma função que recebe um número x e devolve o seu valor absoluto:

```
def abs(x):
    if x >= 0:
      return x
    else:
      return -x
```

Funções

Para especificar que você quer definir uma função você deve utilizar a palavra reservada '**def**'. O que vem a seguir, '**abs**',é o que chamamos de **nome da função**. Tecnicamente, é um identificador. A seguir vem uma lista de nomes. No exemplo acima, há somente um nome, '**x**'. O ':', como de hábito, indica uma relação de subordinação entre os comandos.

```
def abs(x):
  if x >= 0:
    return x
  else:
    return –x
```

Esta sequência, uma vez que a 'x' seja associado um objeto, produzirá o valor absoluto do objeto (que deve ser de tipo numérico).

Como chamar uma função?

Chamar uma função estabelece que queremos executar o código que a ela está associado. Por exemplo,

```
>>> abs(-5)
5
```

A chamada em nosso exemplo tem a forma: abs(<expr>).

A execução ocorre da seguinte forma:

1. <expr> é avaliada produzindo um objeto, digamos v.
2. O objeto v é associado ao nome x, **cujo escopo é a função abs**, que é o único argumento de abs.
3. O código subordinado à abs é então executado. Note que há uma nova palavra reservada, return. Não traduza 'return' como 'retorne'. Traduza-a como 'devolva'.

Funções

O comando return <expr> funciona da seguinte forma:

1. <expr> é avaliada produzindo um objeto, digamos v.
2. 2. A execução da função é interrompida e é devolvido o valor v para o chamador da função.

```
def abs(x):
    if x >= 0:
        return x
    else:
        return –x
```

No exemplo acima, com a chamada abs(-5), ao avaliar o comando if-else a condição x >= 0 se torna -5 >= 0, que é falsa, e portanto, o comando subordinado ao else, 'return -x', é selecionado para execução.

A expressão -x é avaliada produzindo o objeto 5.Finalmente a execução da função é interrompida e o objeto 5 é devolvido ao chamador.

Funções

Ainda no exemplo :

```
def abs(x):
    if x >= 0:
        return x
    else:
        return –x
```

Você pode pensar que o valor 5 substitui a expressão 'abs(-5)' após o término da chamada 'abs(-5)'.É claro que podemos escrever expressões mais complicadas envolvendo funções:

```
>>> abs(10) + abs(-5)
15
```

A execução procede da esquerda para a direita. Assim, primeiro é realizada a chamada abs(10) que produz o objeto 10.A seguir é reavaliada a chamada abs(-5) que produz o objeto 5.Finalmente os números 10 e 5 são somados produzindo o objeto 15. Podemos esquematizar esta computação da seguinte forma:
abs(10) + abs(-5) -> 10 + abs(-5) -> 10 + 5 -> 15.

Uma função é um objeto. Logo podemos associar-lhe um outro nome, se assim desejarmos:

```
>>> modulo = abs
>>> modulo(7)
7
```

Desta forma, escrever modulo(7) é o mesmo que escrever abs(7).

Eis uma função que recebe objetos x e y e devolve um máximo entre x e y(desde que a operação de comparação faça sentido): def max(x, y):

```
  if x >= y:
     return x
   else:
     return y
```

A função **main** se responsabiliza por requisitar os dados de entrada e, depois, chama a função **max**:

```
def main():
  a = int(input('Entre com um número: '))
  b = int(input('Entre com um número: '))
  print(max(a, b))
```

É comum ver a função max, ou uma similar, escrita da seguinte forma:def max(x, y): return x if x >= y else y

Funções

Para que se possa apreciar a forma como o Python lida com funções, considere o programa:

```
def min(x, y):    return x if x <= y else y
def max(x, y):    return x if x >= y else y
def main():
  fs = [min, max]
  x = int(input('Entre com um número: '))
  y = int(input('Entre com outro número: '))
  for m in fs:
    print(m(x, y))
```

Observe que o programa possui três funções (min, max e main). As funções min e max possuem dois argumentos formais, a saber, x e y. A função main não possui argumentos formais.

Ainda sobre o programa do slide anterior

Ao chamarmos uma função escrevendo, por exemplo, max(10, 20), dizemos que o 10 e o 20 são os argumentos reais da função.

Funções são objetos em Python. Logo, podemos manipulá-las de forma similar ao que fazemos com outros objetos. O nome da função nomeia o objeto. A primeira linha da função main cria uma lista fs. Os elementos da lista são as funções min e max. No laço for m in fs: print(m(x, y)) iteramos sobre os elementos de fs, que são as funções min e max. Na primeira iteração m nomeia a função min, e na segunda m nomeia a função max.

Na primeira iteração, como m nomeia min, então a sentença m(x, y) é o mesmo que min(x, y). Analogamente, na segunda iteração, a sentença m(x, y) é o mesmo que max(x, y).

Para executar o programa do slide anterior, devemos chamar a função main:

>>>main()

Funções

Uma função é definida escrevendo-se

def <fun_name>(<arg1>, ..., <argn>):

    <cmds>

1. A definição acima cria um objeto do tipo função nomeado por <fun_name>.

Tal objeto é passivo. Para que a execução seja ativada é necessária uma chamada para a função.

2. <arg_1>, ..., <argn> são nomes e constituem os argumentos formais da função. Os nomes usados têm validade e visibilidade no corpo da função.Uma chamada para a função tem a forma: <fun_name>(<e1>, ..., <en>)em que <e1>, ..., <en> são expressões que quando avaliadas produzem objetos, digamos obj1, ..., objn, que são os argumentos reais da função. O nomes <arg1>, ..., <argn> nomeiam os objeto obj1, ..., objn.

continua no próximo slide

3. Assim que a associação de nomes aos objetos está pronta, os comandos <cmds> são executados.

4. Uma função pode possuir 0 ou mais comandos return <expr>.

Tal comando é avaliado da seguinte forma: <expr> é avaliada produzindo um objeto obj. Em seguida, a execução da função é terminada e o objeto obj é devolvido para o chamador da função. Você pode e deve pensar que a chamada será substituída pelo objeto obj, e a execução seguirá a partir daí da maneira usual.

5. Uma vez que uma função termina sua execução, as variáveis locais utilizadas por ela não têm mais existência.

6. O "tempo de vida" de uma chamada de uma função se inicia na chamada e termina assim que um comando return, nela contido, é executado, ou quando o último comando nela contido é executado.

7. Vale ressaltar que toda função em Python devolve algo. Em particular, se um valor não estiver explícito em seu código, a função devolverá implicitamente o objeto None.

Mutáveis e Imutáveis

Como vimos, os programas em Python manipulam entidades que chamamos de **objetos**. Cada objeto possui um tipo, um valor e uma identidade. O tipo estabelece o conjunto de valores que um objeto pode admitir, além das operações que sobre ele podem ser realizadas.

Mutáveis e Imutáveis

A identidade de um objeto é constante durante o seu tempo de vida. Cada objeto é mutável ou imutável. Um objeto é mutável quando seu valor pode mudar, e, imutável, quando seu valor não pode mudar. Listas, dicionários e conjuntos são mutáveis (há outros objetos mutáveis). Inteiros, números de ponto flutuante, strings e tuplas são imutáveis.

Mutáveis e Imutáveis

Tuplas, strings e listas são sequências. Sequências possuem as seguintes operações:

1) o operador **in** de pertinência que dado um elemento e uma sequência devolve True se e só se o elemento faz parte da sequência,
2) a função len que devolve o tamanho da sequência,
3) operações de fatiamento
4) são iteráveis (podemos percorrê-los elemento a elemento)

Mutáveis e Imutáveis

Suponha que **s** é uma sequência. Cada inteiro i no intervalo 0, 1, ..., len(s)-1é chamado de **índice** (ou posição) de s; s[i] é chamado de **item** de s e nomeia o objeto da posição i.

Listas, como são mutáveis, permitem realizar operações de atribuição nos seus itens.

Assim, é permitido escrever

s[i] = e para uma lista s, um índice i de s, e um objeto e, de tal forma que, após a execução deste comando o item s[i] nomeia o objeto e.

Mutáveis e Imutáveis

Tuplas, como são imutáveis, não permitem realizar operações de atribuição nos seus itens. Logo, não é permitido escrever t[i] = e para uma tupla t, um índice i de t, e um objeto e.

Sugestão: ao usar tuplas certifique-se de que os elementos da tupla também são imutáveis.Isto não é obrigatório, como o exemplo abaixo mostra:

>>> t = (1, [2, 3, 4, 5], 8)

>>> t[1].append(6)

>>> t(1, [2, 3, 4, 5, 6], 8)

Note que no comando t[1].append(6) não ocorre uma atribuição ao item t[1].Há um comportamento inesperado e surpreendente em Python. veja-o:

>>> t = (1, [2, 3, 4, 5], 8)

>>> t[1] = t[1] + [6]

Traceback (most recent call last): File "<pyshell>", line 1, in <module>TypeError: 'tuple' object does not support item assignment

>>> t(1, [2, 3, 4, 5], 8)

Isto ilustra duas coisas:

1) t[1] = t[1] + [6] não equivale a t[1] += [6]

2) t[1] += [6] não equivale a t[1].append(6).

Cuidado! A sugestão é construir suas estruturas de dados de tal forma que em toda tupla todos os seus componentes também sejam imutáveis!

Mais sobre listas

As listas são provavelmente a estrutura de dados mais utilizada em programas em Python. Vamos destacar algumas operações sobre listas. Para algumas delas vamos (ao longo do curso) implementar nossas próprias versões. Como já vimos, para uma lista xs, o método **append** adiciona um novo elemento em xs.

```
>>> xs = [1, 2, 3, 6]
>>> xs.append(5)
>>> xs
[1, 2, 3, 6, 5]
```

Por outro lado, o método pop remove e devolve o último elemento da lista.

```
>>> xs.pop()
5
>>> xs
[1, 2, 3, 6]
```

Mais sobre listas

Para uma lista xs e um objeto x, xs.count(x) devolve o número de ocorrrências de x em xs.

```
>>> xs = [1, 4, 5, 4, 3, 4, 1]
>>> xs.count(4)
3
```

Para uma lista xs, xs.reverse() inverte os elementos de xs.

```
>>> xs = [1, 2, 4, 8]
>>> xs.reverse()
>>> xs
[8, 4, 2, 1]
```

Mais sobre listas

Ainda para uma lista xs, xs.sort() rearranja os elementos de xs em ordem não-decrescente.

```
>>> xs = [4, 6, 1, 2, 7, 8, 3, 5]

>>> xs.sort()

>>> xs

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

Podemos também usar a função sorted que dada uma lista xs produz uma novalista que é um rearranjo em ordem não-decrescente de xs.

```
>>> xs = [4, 6, 1, 2, 7, 8, 3, 5]

>>> ys = sorted(xs)

>>> ys

[1, 2, 3, 4, 5, 6, 7, 8]>>>

 xs

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

Mais sobre listas

Para ordenar xs em ordem não-crescente é só escrever xs.sort(reverse = True).

```
>>> xs = [4, 6, 1, 2, 7, 8, 3, 5]

>>> xs.sort(reverse = True)

>>> xs

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

De maneira similar, podemos produzir uma nova lista em ordem não-crescente.

```
>>> xs = [4, 6, 1, 2, 7, 8, 3, 5]

>>> ys = sorted(xs, reverse = True)

>>> ys

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

Há mais uma série de métodos e funções fundamentais que operam sobre listas que veremos posteriormente.

Módulo 04

COMECE A PROGRAMAR COM PYTHON

Sobre o Módulo 3

Revisão

Funções

Mutáveis e imutáveis

Mais sobre listas

Ordenação

Algoritmos de ordenação são muito importantes na computação. Basicamente ordenam um conjunto de objetos quando a relação de ordem entre eles é possível (determinar se um objeto é maior, menor ou igual a outro, por determinado critério).

Os algoritmos de ordenação são usados diretamente, quando só precisamos mesmo ordenar um conjunto de objetos ou indiretamente, quando a ordenação facilita uma tarefa maior (busca, por exemplo).

Há algoritmos de ordenação mais ou menos eficientes em termos de número de operações necessárias para efetuar a ordenação. Para ordenar um número muito grande de objetos, a eficiência do algoritmo é muito importante.

Ordenação por seleção

Um algoritmo popular, mas não muito eficiente, para ordenação de uma lista é o algoritmo de ordenação por seleção.

Suponha que você queira ordenar a lista

[4, 5, 1, 2, 7, 6, 3].

Para tornar as ideias mais claras vamos exibir a lista separando-a em duas partes por uma barra |.

No início temos | 4 5 1 2 7 6 3.Na primeira iteração, selecionamos um menor elemento dentre os que ocorrem depois da |; no caso, é o número 1.

Naturalmente, o 1 deve ser o primeiro elemento da lista.

Assim, vamos trocar o 1 com o elemento que ocorre imediatamente depois da barra,

1 | 5 4 2 7 6 3.

Note que movemos a | para frente.

Isto reflete uma parte do conhecimento acumulado com o processamento. Sabemos que o que vem antes da barra está ordenado, e, além disso, se tomarmos qualquer par de elementos tal que um deles, digamos x, ocorre antes da barra e o outro, digamos y, ocorre depois da barra, então x <= y.

Vamos à segunda iteração. A ideia é repetir o mesmo procedimento que fizemos anteriormente. Agora, vamos selecionar um menor elemento dentre os que ocorrem depois da barra; no caso, é o número 2. Feito isso, em seguida, trocamos o elemento que ocorre imediatamente após a barra com o 2, obtendo:

1 2 | 4 5 7 6 3.

Ao movermos a barra adiante, notamos que os elementos que vêm antes da barra estão ordenados. Além disso, se x ocorre antes da barra e y depois da barra, então x <= y. Repetindo-se sucessivamente estas operações, obtemos: 1 2 3 | 5 7 6 4, (aqui troca-se o 4 com o 3!)

Ordenação por seleção

Continuando...

1 2 3 4 | 7 6 5, (aqui troca-se o 5 com o 4!)

1 2 3 4 5 | 7 6,

1 2 3 4 5 6 | 7.

Note que quando resta só um elemento depois da barra não é necessário realizar nenhum processamento, pois ele é maior ou igual a qualquer elemento antes da barra.

Temos que pensar como produzir um programa em Python para implementar esta ideia. Precisamos realizar duas operações fundamentais:

1) Encontrar um menor elemento de um sufixo da lista (uma parte final da lista), e
2) Trocar o menor elemento de posição com o elemento inicial deste sufixo.

Ordenação por seleção

Ora, para realizar as operações 1) e 2) vamos precisar determinar o índice de um menor elemento de um sufixo para poder trocá-lo com o elemento da posição do início do sufixo. Concretamente, na primeira iteração vamos encontrar um índice de um menor elemento do sufixo de [4, 5, 1, 2, 7, 6, 3] que começa no índice 0.Na segunda iteração vamos encontrar um índice de um menor elemento do sufixo de [1, 5, 4, 2, 7, 6, 3] que começa na posição 1 (justamente onde está a barra!!!). Na terceira iteração vamos encontrar o índice de um menor elemento do sufixo de [1, 2, 4, 5, 7, 6, 3] que começa na posição 2.

Esta operação pode ser abstraída na seguinte função que recebe uma lista xs e um índice i tal que i está no intervalo 0, 1, len(xs)-1 e devolve o índice de um menor elemento de xs[i:] (ou seja, a partir do índice i).

```
def indmin(xs, i):
    m = i
    for k in range(i+1, len(xs)):
        if xs[k] < xs[m]: m = k
    return m
```

Ordenação por seleção

Com esta função (indmin) , finalmente podemos escrever o algoritmo de ordenação por seleção:

```
def indmin(xs, i):
    m = i
    for k in range(i+1, len(xs)):
        if xs[k] < xs[m]: m = k
    return m


def ssort(xs):
    for i in range(len(xs)-1):
        m = indmin(xs, i)
        xs[m], xs[i] = xs[i], xs[m]
```

Exemplo de uso :

```
>>> ls = [1,4,2,7,8,3,5,9]
>>> ssort(ls)
>>> ls
[1, 2, 3, 4, 5, 7, 8, 9]
```

Ordenação por seleção

Este primeiro algoritmo de ordenação (selection sort) ilustra um processo típico em programação que consiste em decompor um problema em subproblemas e usar as soluções para os **subproblemas** para compor uma solução para o problema original. Tipicamente esta decomposição dá origem a **funções** que resolvem estes subproblemas.

Ordenação por Inserção

Vamos às preliminares para entender a **ordenação por inserção**

Uma lista xs **está ordenada** se

xs[i-1] <= xs[i] para cada i em [:len(xs)].

Dizemos que um par de índices i e k é uma inversão em xs se

i < k e xs[i] > xs[k].

Um forma natural de ordenar uma lista consiste em determinar uma inversão i, k e trocar os conteúdos das posições i e k, ou seja, realizar a operação xs[i], xs[k] = xs[k], xs[i].Ora, se xs tem pelo menos uma inversão, então existe i em [:len(xs)] tal que i-1, i é uma inversão. Observe o seguinte exemplo:

Xs=[2 4 6 3 7 9 5]

O par 2, 3 é uma inversão, pois xs[2] > xs[3] (ou seja, 6>3).Isto sugere o seguinte algoritmo: encontre (se existir) um índice i tal que i-1, i é uma inversão; se um tal par não existir, então pare, senão troque os elementos das posições i-1 e i.

Ordenação por Inserção

O primeiro subproblema a resolver consiste em encontrar, se existir, um índice i tal que i-1, i é uma inversão. A seguinte função resolve este problema:

```
# recebe uma lista xs e devolve len(xs) se xs estiver ordenada,
# caso contrário, devolve i em [1:len(xs)] tal que i-1, i é uma inversão.
def findinv(xs):
    i = 1
    while i < len(xs) and xs[i-1] <= xs[i]: i+= 1
    return i
```

Note que há duas formas do laço while terminar:

1) i == len(xs). Neste caso, xs[i-1] <= xs[i] para cada i em [1:len(xs)] e, portanto, xs está ordenada.
2) 2) i < len(xs) e xs[i-1] > xs[i].

Ordenação por Inserção

Eis a implementação algoritmo de ordenação acima descrito, que utiliza a função findinv do slide anterior

```
def invsort(xs):
    while True:
        i = findinv(xs)
        if i == len(xs): return
        xs[i], xs[i-1] = xs[i-1], xs[i]
```

O algoritmo de ordenação por inserção pode ser encarado como uma versão mais inteligente do invsort.

Ordenação por Inserção

Vamos considerar primeiro o seguinte subproblema:

dado uma lista xs e um índice i em [:len(xs)] tal que xs[:i] (ou seja, do índice 0 até i-1) está ordenada, rearranjar (ou seja, trocar de posição) os elementos de xs[:i+1] de tal forma que xs[:i+1] fique ordenada. Você pode dizer isso informalmente afirmando que o algoritmo insere o elementoxs[i] na posição correta para garantir que xs[:i+1] fique ordenado. Eis um exemplo:

i=4 xs=[2 5 7 9 10 6]

Note que os dados acima satisfazem a hipótese previamente destacada: xs[:i] está ordenada.

A ideia agora consiste em comparar elementos adjacentes e trocá-los caso seja necessário. Ora, 6 < 10, donde é necessário inverter a ordem destes elementos. Obtemos assim a lista 2 5 7 9 6 10.O elemento 6 é o que desejamos levar para a posição correta.

Agora, vamos então comparar este elemento com o seu novo predecessor, no caso, o 9. Como 6 é menor que 9, mais uma troca é necessária e a lista 2 5 7 6 9 10 é obtida.

O novo predecessor do 6 é o 7 e, mais uma vez, uma troca é compulsória. Tal troca produz a lista 2 5 6 7 9 10.

Finalmente, o processo pára uma vez que 6 é maior que 5.

```
def insert(xs, i):
    k = i
    while k > 0 and xs[k] < xs[k-1]:
        xs[k], xs[k-1] = xs[k-1], xs[k]
        k -= 1
```

É importante ressaltar que o compromisso da função é produzir um rearranjo ordenado de xs[:i+1] desde que xs[:i] esteja ordenado por hipótese.

Na condição que controla o laço 'k > 0 and xs[k] < xs[k-1]'a ordem dos termos da conjunção é violentamente importante. A validade do primeiro termo 'k > 0' garante que o segundo termo ’xs[k] < xs[k-1]' faz sentido. Note também o papel da variável k. O valor de k durante cada iteração do laço é a posição do elemento que queremos inserir.

Ordenação por Inserção

Eis a função que implementa o algoritmo de **ordenação por inserção**:

```
def isort(xs):
    for i in range(1, len(xs)):
        insert(xs, i)
```

Note que a primeira iteração começa com i valendo 1,pois é claro que xs[:1] está ordenada. A chamada insert(xs, 1) rearranja xs[:2] de tal forma que xs[:2] fique ordenada.

Na próxima iteração i vale 2.Ora, xs[0:2] está ordenada, donde a chamada insert(xs, 2) produz um rearranjo ordenado de xs[0:3].

Generalizando o argumento, fica fácil ver que a função devolve um rearranjo ordenado de xs[:len(xs)], como queríamos.

Busca Linear

Se xs é uma lista e x é um objeto, então x in xs devolve True se, e só se, x é um dos elementos de xs. Abaixo segue a implementação de um algoritmo, conhecido como busca linear, que essencialmente é o que é realizado internamente quando você escreve uma expressão da forma x in xs quando xs é uma lista.

```
def lsearch(x, xs):
    for i in range(len(xs)):
        if xs[i] == x: return True
    return False
```

Busca Linear

Observe que se x não está em xs, então é necessário percorrer toda a lista para que isto seja certificado. Assim, são necessárias, neste caso, len(xs) comparações da forma 'xs[i] == x' para decidir a pertinência de x em xs. Será que podemos fazer melhor?

Busca Binária

Podemos fazer algo melhor que a busca linear?

Para responder a pergunta anterior, será necessário adicionar uma hipótese sobre a lista xs. Vamos, no que segue, **admitir que xs está ordenada**, ou seja,xs[0] <= xs[1] <= ... <= xs[len(xs)-1]. Para fixar as ideias, vamos considerar o seguinte exemplo:

x=[2 5 7 9 10 12 16]. É claro que se x está em xs, então x está em xs[0:7].Vamos chamar o intervalo [0:7] de espaço de busca.

Digamos que o elemento a ser buscado seja o 12. Escolha um índice qualquer no intervalo [0:7]. Por exemplo, o índice 3. Observe que xs[3] == 9 < 12.Isto implica se o 12 está em xs[0:7], então12 obrigatoriamente está em xs[4:7]. Note que o espaço de busca “encolheu” de [0:7] para [4:7].

Vamos repetir o argumento com o espaço de busca [4:7].Selecione um índice qualquer em [4:7], digamos 6.Ora, xs[6] == 16 > 12, donde se 12 é um elemento de xs[4:7], então 12 deve ser também um elemento de xs[4:6].Mais uma vez, o espaço de busca encolheu, desta vez de [4:7] para [4:6]. Agora, selecione um índice em [4:6], digamos 5.

Como xs[5] == 12, então o algoritmo acabou de encontrar o elemento procurado!

Busca Binária

O que ocorreria se o elemento a ser buscado fosse o 11?Neste último passo, o espaço de busca se reduziria para [4:5]. Agora, há um único elemento, o 4, em [4:5].Como xs[4] == 10 < 11, então se 11 está em xs[4:5],então 11 está em xs[5:5]. O espaço de busca agora se reduziu para [5:5] que tem tamanho 0. Logo, 11 não está em xs[0:7].

Uma implementação deste algoritmo deve decidir como selecionar um índice m no espaço de busca [e:d].Vamos fazer isso, selecionando m como o índice (e+d)//2.Isto implica que o espaço de busca é dividido por 2 a cada iteração.

Por exemplo, se o espaço de busca tiver tamanho 1.000.000, então em não mais que 20 iterações é possível decidir se o elemento está presente na lista. Este algoritmo pode ser implementado da seguinte forma:

```
def bsearch(x, xs):
    e, d = 0, len(xs)
    while e < d:
        m = (e+d) // 2
        if x == xs[m]: return True
        elif x < xs[m]: d = m
        else: e = m+1
        return False
```

Obs: obviamente, a lista xs deve estar ordenada

Módulo 05

COMECE A PROGRAMAR COM PYTHON

Sobre o Módulo 4

Revisão

Ordenação por seleção

Ordenação por inserção

Busca Binária

Funções como objetos

Em Python, funções são objetos de primeira classe. Isto quer dizer que :

(1) variáveis podem nomear funções(além, é claro, do nome dado por definição),

(2) funções podem ser armazenadas em listas e outras estruturas de dados,

(3) podem ser argumentos reais de outras funções,

e

(4) podem ser devolvidas por outras funções.

Funções como objetos

funcoesobj.py

```
def maplist(f, xs):
    return [f(x) for x in xs]

def double(x): return 2*x
```

Shell

```
Python 3.6.1
>>> %Run funcoesobj.py
>>> maplist(double, [1, 2, 3, 4])

[2, 4, 6, 8]
```

Exemplo :
A função maplist(f, xs) recebe uma função f e uma lista xs de objetos do tipo A, e devolve a lista [f(x1),...,f(xn)] onde xs é a lista [x1,..., xn].

Observe o exemplo acima. A chamada maplist(double, [1, 2, 3,4]) devolve a lista [double(1), double(2), double(3), double(4)]que é a lista [2, 4, 6, 8].

Funções como objetos

É evidente que uma função como maplist, em virtude das list comprehensions, se torna um tanto quanto desnecessária. No entanto, é importante ressaltar que funções são objetos que podem ser manipulados como outros objetos da linguagem. Observação: O Python 3 possui uma função map (**cuidado**, em Python 2, o comportamento desta função é diferente).

Funções como objetos

Considere o exemplo de “map” :

```
>>> es = map(double, [1, 2, 3, 4])
>>> es
<map object at 0x034E9DB0>
>>> list(es)
[2, 4, 6, 8]
```

A chamada a map devolve o que se chama de uma expressão geradora (logo, não é uma lista). Se você quiser construir uma lista com o resultado de um map, você deve fazê-lo explicitamente como ilustrado acima.

Lambdas (funções anônimas)

lambda é uma palavra reservada do Python que é utilizada para criar uma função anônima. A sintaxe é a seguinte

lambda <parametros>: <expressão>

o corpo de um lambda que é a entidade sintática <expressão> não pode conter atribuições e nem comandos como while, for, etc. Vamos a um exemplo:

```
>>> soma = lambda x, y: x + y
>>> soma(10,20)
30
>>> soma('aa', 'bb')

'aabb'
```

A primeira linha, no lado direito da atribuição, cria uma função anônima que recebe dois objetos x e y e devolve x + y. Note que a operação **+** depende da natureza dos argumentos reais. A atribuição associa o nome soma à função anônima criada pelo lambda.

Lambdas (funções anônimas)

A função sorted pode receber como argumento um lambda no parâmetro nomeado por key. Isto determinará como a função ordenará a lista de entrada, como ilustrado no que segue:

```
>>> xs = [(1, 2), (2, 3), (1, 10), (2, 1), (5, 6)]
>>> ys=sorted(xs)
>>> ys
[(1, 2), (1, 10), (2, 1), (2, 3), (5, 6)]
```

Como é de se esperar, a função ordena as duplas primeiro de acordo com o primeiro elemento da tupla e depois de acordo com o segundo elemento da tupla. Ora, se quisermos ordenar, por exemplo, pela soma dos componentes da dupla podemos fazê-lo da seguinte forma:

```
>>> zs = sorted(xs, key = lambda t: t[0] + t[1])
>>> zs
[(1, 2), (2, 1), (2, 3), (1, 10), (5, 6)]
```

Lambdas (funções anônimas)

O paramêtro key (que quer dizer chave) é uma função aplicada a cada elemento da lista antes de compará-los. No nosso exemplo, key é a função que soma os componentes da dupla. Assim, na nova relação de ordem (2, 3) é menor que (1, 10), pois 2 + 3 é menor que 1 + 10. Eis mais um exemplo no mesmo espírito:

```
>>> xs = [(10, 5), (4, 6), (8, 9), (1, 20), (3, 12)]
>>> min(xs, key = lambda t: t[0] + t[1])
(4, 6)
```

Pense sobre este exemplo:

```
>>> soma = lambda x: lambda y: x + y
>>> soma5 = soma(5)
>>> soma5(10)
15
```

Vamos a um outro uso mais sofisticado do lambda.

Um conjunto em matemática é uma entidade que contém elementos. Conjuntos podem ser unidos. A união de dois conjuntos X e Y produz um novo conjunto cujos elementos são os elementos de X e Y, ou seja, um elemento está na união se ele está em X ou está em Y. O conjunto vazio é um conjunto que não contém elementos. Vamos representar conjuntos em Python usando lambdas.

A nossa representação é para ilustração. Ela contém alguns deslizes. Será que você é capaz de encontrá-los? Um conjunto na nossa implementação é uma função que recebe um objeto e devolve um booleano. Atente para o fato de que vamos utilizar o termo 'conjunto' tanto no sentido matemático quanto no sentido da implementação que vamos fazer em Python

O conjunto vazio pode ser representado pela função empty.

```
def empty():
      return lambda x: False
```

A função empty devolve uma função que recebe um elemento x e devolve False.

```
>>> e=empty()
>>> e(2)
False
>>> e(1)
False
```

Em geral, e(obj) é False para qualquer objeto obj. Vamos agora escrever uma função para representar um conjunto unitário, isto é, um conjunto com um único elemento.

```
def singleton(x):
    return lambda u: x == u
```

Lambdas (funções anônimas)

A função singleton (slide anterior) recebe um objeto x e devolve uma função que quando recebe um objeto u devolve verdadeiro se e somente se u e x são iguais.
A função abaixo implementa a união de dois conjuntos:

```
def union(s, t):
    return lambda u: s(u) or t(u)
```

A função union recebe conjuntos s e t e devolve uma função que, dado um objeto u, devolve True se e só se u está no conjunto s ou no conjunto t.

Lambdas (funções anônimas)

A fim de ilustração, eis uma função que recebe uma lista xs e devolve um conjunto cujos elementos são os objetos de xs.

```
def newset(xs):
    s = empty()
    for x in xs:
        t = singleton(x)
        s = union(s, t)
    return s
```

```
>>> s = newset([1, 2, 3, 4, 10, 8])
>>> s(1)
True
>>> s(4)
True
>>> s(5)

False
>>> s(10)
True
```

Expressões Geradoras

Considere a seguinte list comprehension:

```
>>> xs = [2*i for i in range(10)]
>>> xs
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

A seguinte construção, que substitui os colchetes por parênteses, é conhecida como uma expressão geradora.

```
>>> gs = (2*i for i in range(10))
>>> gs
<generator object <genexpr> at 0x02DBACF0>
```

Podemos construir uma lista a partir de uma expressão geradora:

```
>>> list(gs)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Note, entretanto, que uma expressão geradora não é uma lista.

Expressões Geradoras

A escolha entre usar uma list comprehension ou uma expressão geradora é delicada. Uma list comprehension evidentemente constrói uma lista. Portanto, devemos sempre levar em conta o espaço que tal lista pode ocupar. Uma expressão geradora, por outro lado, constitui uma forma de gerar um a um os itens para processamento. Vamos exemplificar. Quando se escreve **for elmt in xs: print(elmt)** a lista **xs** está construída e o laço faz com que **elmt** nomeie um a um os elementos de xs.

Expressões Geradoras

Quando se escreve **for elmt in gs: print(elmt)** o efeito será o mesmo, ou seja, os primeiros dez números pares serão escritos na saída padrão.No entanto, não é criada nenhuma lista para isso. O processamento ocorre da seguinte forma:

-- no início i vale 0 e o objeto gerador produz o valor 0 que elmt nomeia;

-- o valor 0 é escrito na saída padrão;

-- o laço volta a ser executado, agora com o próximo valor do range que é o 1 (nomeado por i), e o objeto gerador produz o valor 2 (em virtude da expressão 2*i) que elmt agora nomeia;

continua

Expressões Geradoras

-- o laço volta a ser executado, agora com o próximo valor do range que é o 2 (nomeado por i), e o objeto gerador produz o valor 4 (em virtude da expressão 2*i) que elmt agora nomeia;

-- o valor 4 é escrito na saída padrão; Isto se repete até que o range produza o valor 3, e o objeto gerador produza o valor 6 que enfim é escrito na saída padrão. Note que em tal processamento nenhuma lista está envolvida.

Observe o exemplo:

```
>>> gs = (2*i for i in range(4))
>>> for elmt in gs: print(elmt)


0
2
4
6

>>> for elmt in gs: print(elmt)

>>>
```

Note que nada ocorreu na segunda vez em que requisitamos a impressão dos elementos, pois todos os elementos já foram consumidos, ou seja, o objeto gerador não tem mais elementos para produzir.

Expressões Geradoras

Considere agora o seguinte exemplo:

```
>>> gs = (2*i for i in range(10))
>>> for elmt in gs:
        if elmt > 10: break
        print(elmt)


0
2
4
6
8
10
>>> for elmt in gs: print(elmt)


14
16
18
```

O comado break interrompe a execução do laço mais interno que o contém e vai para comando que segue tal laço. Assim, no exemplo acima os números 0, 2, 4, 6, 8, e 10 são escritos na saída padrão, pois são todos menores ou iguais a 10.Quando o objeto gerador produz o valor 12, o laço é interrompido. Ora, como o exemplo ilustra, o laço
for elmt in gs: print(elmt)
começa gerando o próximo número, o 14 (lembre-se que o primeiro processamento foi interrompido no valor 12).

Expressões Geradoras

A vantagem da utilização de expressões geradoras reside no menor consumo de memória e no fato de que, quando comparado a uma list comprehension, esta última exige a construção da lista. Quando a quantidade de elementos é muito grande, e o processamento envolvendo tais elementos consiste em realizar alguma operação e depois decartá-los é comum optar-se por uma expressão geradora. No entanto, há um gasto adicional de processamento na implementação dos objetos geradores. Logo, a escolha pode não ser tão simples.

Módulo 06

COMECE A PROGRAMAR COM PYTHON

Sobre o Módulo 5

Revisão

Funções como Objetos

Lambdas (funções anônimas)

Expressões geradoras

Tuplas

Tuplas, listas e strings são sequências, como já vimos. Uma **tupla** é imutável. Assim, se t é uma tupla e i está em [:len(t)]então a operação de atribuição t[i] = ... não está definida; é um erro. Podemos fazer fatiamentos sobre tuplas, que geram novas tuplas. Para ordenar uma tupla, existem duas alternativas:(1) Converta a tupla em uma lista e ordene a lista. Por exemplo :

```
>>> t = (4, 3, 1, 2)
    xs=list(t)
>>> xs
[4, 3, 1, 2]
>>> xs.sort()
>>> xs
[1, 2, 3, 4]
```

(2) Utilize a função sorted.

```
>>> t = (4, 3, 1, 2)
>>> xs=sorted(t)
>>> xs
[1, 2, 3, 4]
```

Note que sorted devolve uma lista.

Tuplas

Lembre-se que conjuntos não podem conter listas como elementos devido a mutabilidade das listas.

```
>>> s = { [1, 2, 4], [5, 2, 3], [7] }
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: unhashable type: 'list'
```

No entanto, conjuntos podem conter **tuplas** como elementos (observe que uma tupla com um único objeto x é escrita (x,)) :

```
>>> s={(7,), (5, 2, 3), (1, 2, 4)}
>>> s
{(7,), (5, 2, 3), (1, 2, 4)}
```

Alguns métodos são comuns tanto a listas quanto a tuplas. Por exemplo, para uma tupla t e um objeto x, t.count(x) devolve o número de ocorrências de x em t. Por outro lado, t.index(x) devolve, se existir, o índice da primeira ocorrência de x em t; caso uma tal ocorrência não exista, o método levanta uma exceção. É claro, você também pode utilizar o operador in de pertinência com tuplas, assim como é possível fazê-lo com as demais sequências.

Conjuntos

Um tipo **conjunto** em Python possui duas variedades: a mutável, dada pelo tipo set, e a imutável, dada pelo tipo frozenset(). Conjuntos são iteráveis. Assim, se **s** é um conjunto, então é possível iterar sobre seus elementos escrevendo, por exemplo, **for e in s.**

Um conjunto **s** também é dotado da função **len** que devolve o número de elementos do conjunto, e da operação de pertinência, **in**, que dado um objeto e um conjunto, devolve True *se e só se* o objeto é um elemento do conjunto.

Conjuntos

Vamos a alguns exemplos:

```
>>> s = { 2, 4, 1, 5, 6, 10 }
>>> for e in s: print(e)


 1
 2
 4
 5
 6
 10
>>> 8 in s
False
>>> 5 in s
True
```

Note que os elementos do conjunto são envolvidos por um { e um } , notação esta que remete a padrão em matemática. Um conjunto vazio pode ser criado escrevendo-se set(). Cuidado, escrever { } produz um dicionário vazio e não um conjunto vazio.

Conjuntos

Lembre-se que com listas também podemos escrever

```
>>> xs = [1, 2, 4, 10, 6, 5]
>>> 4 in xs
True
```

Qual a diferença do operador **in** envolvendo listas e conjuntos? A diferença está na performance. Enquanto que a operação **e in xs** quando xs é uma lista envolve, no pior caso, um número de passos proporcional a len(xs), a operação **e in s** quando s é um conjunto envolve, no caso médio, um número fixo de operações, independente do tamanho do conjunto.

Por outro lado, uma lista possui uma noção de ordem: há o primeiro elemento, o segundo elemento, e assim, sucessivamente. No caso de um conjunto esta noção não existe. Desta forma, se **s** é um conjunto, então não faz sentido escrever s[i]! A discussão precedente sugere que se a operação dominante que você precisa realizar é a de pertinência, então certamente é mais vantajoso utilizar conjuntos em vez de listas. No entanto, se a operação dominante consiste em iterar pelos elementos, então é mais conveniente utilizar listas. Vamos ilustrar algumas operações que podemos realizar sobre conjuntos:

1) adicionar um elemento x a um conjunto s: s.add(x) :

```
>>> s={1, 2, 4, 5, 6, 10}
>>> s.add(7)
>>> s
{1, 2, 4, 5, 6, 7, 10}
>>> s.add(4)
>>> s
{1, 2, 4, 5, 6, 7, 10}
```

Note que, como é de se esperar, um conjunto não mantém cópias de elementos.

Conjuntos

Podemos também remover um elemento x de um conjunto s: s.discard(x).

```
>>> s
{1, 2, 4, 5, 6, 7, 10}
>>> s.discard(2)
>>> s
{1, 4, 5, 6, 7, 10}
>>> s.discard(9)
>>> s
{1, 4, 5, 6, 7, 10}
```

Note que nada ocorre se o elemento a ser removido não está presente no conjunto.

A união de dois conjuntos s e t pode ser obtida fazendo-se s.union(t) que produz um novo conjunto cujos elementos são os elementos que estão em s ou em t. Isto também pode ser escrito s | t.

```
>>> s
{1, 4, 5, 6, 7, 10}
>>> t={1,5,6,8}
>>> v=s.union(t)
>>> v
{1, 4, 5, 6, 7, 8, 10}
>>> w=s|t
>>> w
{1, 4, 5, 6, 7, 8, 10}
```

Conjuntos

A intersecção de dois conjuntos s e t também pode ser obtida escrevendo-se s.intersection(t), ou ainda, usando-se a forma mais abreviada s & t. Note que a operação gera um novo conjunto.

```
>>> s
{1, 4, 5, 6, 7, 10}
>>> t
{8, 1, 5, 6}
>>> w=s&t
>>> w
{1, 5, 6}
```

Se você quiser remover um elemento arbitrário de um conjunto s, basta escrever s.pop().

```
>>> s
{1, 4, 5, 6, 7, 10}
>>> s.pop()
1
>>> s
{4, 5, 6, 7, 10}
```

É importante salientar que o elemento removido é arbitrário. Não há como prever qual será o elemento removido. Logo, uma operação pop deve ser realizada quando for necessário processar um elemento qualquer do conjunto.

Conjuntos

Podemos também testar se um conjunto **s** é subconjunto de um conjunto t, isto é, se para cada objeto e: e em s implica e em t. Para isso, basta escrever s.issubset(t), ou de forma mais curta, s <= t.

```
>>> s = { 1, 4, 5, 7 }
>>> t = { 2, 1, 7, 8, 4, 5 }
>>> s <= t
True
```

A natureza dos elementos que podem fazer parte de conjuntos deve ser imutável! Assim, não é possível criar um conjunto que contenha uma lista como um de seus elementos.

```
>>> s = { [1, 2, 3], 4, 7 }
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: unhashable type: 'list'
```

No entanto, um conjunto pode conter tuplas como elementos.

```
>>> s = { (1, 2, 4), (5, 6) }
>>> s
{(5, 6), (1, 2, 4)}
```

Set comprehensions

A sintaxe para um set comprehension é similar a de uma list comprehension. A diferença é que a expressão está envolvida em um { e um }:

{ <exp> for <item> in <iterable> }

{ <exp> for <item> in <iterable> if <cond> }

Eis alguns exemplos:

```
>>> xs = [1, 4, 5, 6, 6, 4, 1, 4, 2, 2, 1]
>>> s = { e for e in xs }
>>> s
{1, 2, 4, 5, 6}
>>> t = { e for e in s if e < 5 }
>>> t
{1, 2, 4}
```

Frozen sets

Um frozenset (conjunto congelado) é um conjunto que uma vez criado não pode ser modificado. Portanto, um frozenset é um imutável. Assim, frozenset's podem ser elementos de conjuntos:

```
>>> s = { {1, 2, 3}, {4, 5, 6} }
 Traceback (most recent call last):
   File "<pyshell>", line 1, in <module>
 TypeError: unhashable type: 'set'
>>> s = { frozenset({1, 2, 3}), frozenset({4, 5, 6}) }
>>> s
{frozenset({1, 2, 3}), frozenset({4, 5, 6})}
```

Módulo 07

COMECE A PROGRAMAR COM PYTHON

Sobre o Módulo 6

Revisão

- ✓ Tuplas
- ✓ Conjuntos

Mapeamentos

O conceito de conjunto é fundamental em matemática. A sua importância como ferramental conceitual se torna evidente quando observamos que diversas linguagens de programação, entre elas o Python, fornecem uma representação de conjuntos (finitos).Um outro conceito fundamental em matemática é o de mapeamento ou função. Para evitar confusão com a noção de função em Python, *vamos reservar o termo mapeamento para falarmos sobre as funções da matemática*.

Um par ordenado tem a forma (x, y), em que x e y são elementos tomados de um certo conjunto. Por definição, os pares (x, y) e (z, w) são iguais se e só se x = z e y = w. Um **mapeamento** é um conjunto f de pares ordenados tais que se (x, y) e (x, z) estão em f, então y = z (ou equivalentemente, não existem dois pares no conjunto com o mesmo primeiro elemento). Como é de se esperar, uma das formas de se representar um mapeamento finito consiste em exibir os pares que fazem parte do mapeamento. Por exemplo,

(1) f = { (1, 2), (2, 5), (4, 9), (0, 6) }é um mapeamento.

Note que não existem dois pares distintos com um mesmo primeiro elemento. Por outro lado, o conjunto { (2, 4), (3, 6), (7, 8), (2, 5) } **não é** um mapeamento, em virtude dos pares (2, 4) e (2, 5). Considere o mapeamento (1). É legítimo então escrever f(1) = 2, f(2) = 5, f(4) = 9e f(0) = 6, pois não há dois pares com uma primeira coordenada. Em geral, se f é um mapeamento e (x, y) está em f, então escrevemos f(x) = y. Para um mapeamento f, o domínio de f, escrito dom(f), é o conjunto dos primeiros elementos dos pares de f. Por outro lado, a imagem de f, escrita im(f), é o conjunto dos segundos elementos dos pares de f. Considere, por exemplo, o mapeamento f dado em (1).Então dom(f) = { 0, 1, 2, 4 } e im(f) = { 2, 5, 6, 9 }

Dicionários

Já vimos que uma função em Python se comporta como um mapeamento em matemática. Além disso, listas, sequências e tuplas também se comportam como mapeamentos (é claro, como domínios específicos). A noção de mapeamento em matemática não pode ser capturada por uma outra equivalente em qualquer linguagem de programação, tal restrição está na natureza do processo computacional! **Um dicionário em Python também se comporta como um mapeamento.**

Dicionários

Um **dicionário** é uma coleção **finita** e **não-ordenada** de 0 ou mais pares de objetos da forma chave-valor tal que

(1) o conjunto dos pares do dicionário é um mapeamento, e
(2) as chaves são imutáveis.

Note que os valores podem ser quaisquer objetos. Eis como você pode criar um dicionário vazio:

```
>>> d=dict()
>>> d
{}
>>> d={}
>>> d
{}
```

Dicionários

Suponha que você quer associar alunos a notas. Digamos que os pares sejam: - Jose 7.0, - Maria 8.2, - Godel 10, - Wittgenstein 10, e - Daniel 7.8.

Isto pode ser feito escrevendo-se:

```
>>> d = { 'Jose': 7.0, 'Maria': 8.2, 'Godel': 10, 'Wittgenstein': 10, 'Daniel': 7.8 }
>>> d
{'Jose': 7.0, 'Maria': 8.2, 'Godel': 10, 'Wittgenstein': 10, 'Daniel': 7.8}
```

Dicionários

Podemos alterar de uma forma natural a associação entre uma chave **k** e um valor **v** em um dicionário d, escrevendo-se d[k] = v (que leva tempo constante, no caso médio). Por exemplo :

```
>>> d = { 'Jose': 7.0, 'Maria': 8.2, 'Godel': 10, 'Wittgenstein': 10, 'Daniel': 7.8 }
>>> d['Jose']=8.0
>>> d
{'Jose': 8.0, 'Maria': 8.2, 'Godel': 10, 'Wittgenstein': 10, 'Daniel': 7.8}
```

As chaves de um dicionário d podem ser obtidas escrevendo-se d.keys(); note que d.keys() corresponde ao conceito de domínio em um mapeamento.

```
>>> d.keys()
dict_keys(['Jose', 'Maria', 'Godel', 'Wittgenstein', 'Daniel'])
```

Dicionários

Para testar se uma chave k faz parte de um dicionário d, basta escrever k in d.keys(), ou ainda, k in d. Suponha que d é um dicionário e k é uma chave tal que k não está em d.keys(). Se v é um objeto qualquer, então a operação d[k] = v adicionará o par k-v ao dicionário.

```
>>> d['Chopin'] = 9.5
>>> d
{'Jose': 8.0, 'Maria': 8.2, 'Godel': 10, 'Wittgenstein': 10, 'Daniel': 7.8, 'Chopin': 9.5}
```

É possível também consultar o valor associado a uma certa chave k em um dicionário descrevendo-se d[k], que leva tempo constante (no caso médio). Note, entretanto, que se k não está em d.keys(), então uma exceção KeyError é levantada.

```
>>> d['Newton']
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
KeyError: 'Newton'
```

Dicionários

Se d é um dicionário, então d.items() fornece os pares que fazem parte deste dicionário.

```
>>> d.items()
dict_items([('Jose', 8.0), ('Maria', 8.2), ('Godel', 10), ('Wittgenstein', 10), ('Daniel', 7.
8), ('Chopin', 9.5)])
```

Assim, se você quiser iterar sobre os pares de um dicionário, basta fazer:

```
>>> for it in d.items(): print(it[0] + ': ' + str(it[1]))


Jose: 8.0
Maria: 8.2
Godel: 10
Wittgenstein: 10
Daniel: 7.8
Chopin: 9.5
```

Dicionários

Se você quiser iterar somente sobre as chaves de um dicionário, basta fazer:

```
>>> for k in d.keys(): print(k)


Jose
Maria
Godel
Wittgenstein
Daniel
Chopin
```

ou

```
>>> for k in d: print(k)


Jose
Maria
Godel
Wittgenstein
Daniel
Chopin
```

Dicionários

para iterar sobre os valores de um dicionário, basta fazer:

```
>>> for v in d.values():print(v)


 8.0
 8.2
 10
 10
 7.8
 9.5
```

Como é de se esperar, há uma série de operações que você pode realizar sobre um dicionário. Por exemplo, às vezes, em vez de consultar o valor associado a uma chave k escrevendo-se d[k] (que pode levantar uma exceção),é mais conveniente escrever d.get(k) que devolve None, em vez de levantar uma exceção quando k não está em d.keys().

```
>>> v = d.get('Bohr')
>>> print(v)
 None
```

Dicionários

Os objetos devolvidos pelas chamadas d.items(), d.keys() e d.values() são chamados de visões (views). Como vimos, você pode iterar sobre estes objetos. Há, entretanto, uma peculiaridade, se você modificar o dicionário, a respectiva visão também será modificada.

```
>>> keys = d.keys()
>>> keys
dict_keys(['Jose', 'Maria', 'Godel', 'Wittgenstein', 'Daniel', 'Chopin'])
>>> del(d['Jose'])

>>> keys
dict_keys(['Maria', 'Godel', 'Wittgenstein', 'Daniel', 'Chopin'])
```

Para finalizar esta introdução, vamos resolver um problema utilizando-se dicionários. O problema é o seguinte. É dado um inteiro n >= 1 e uma sequência de n pares da forma objeto - quantidade. O problema consiste em determinar o total de cada objeto. Por exemplo, para a entrada

9
borracha 5
caneta 10
papel 5
caneta 20
papel 7
lapis 9
caneta 10
lapis 9
borracha 5

devemos ter como saída borracha 10 caneta 40 papel 12 lapis 18

Dicionários

Eis um programa em Python que resolve o problema do slide anterior.

```
def read_data():
    data = []
    n = int(input())
    for i in range(n):
        k,v = input().split()
        data.append((k, int(v)))
    return data

def consolidate(xs):
    d = { }
    for (k, v) in xs:
        if k in d: d[k] += v
        else: d[k] = v
    return d

def main():
    data = read_data()
    print(consolidate(data))
```

Observe que não é necessário armazenar os dados em uma lista. Você poderia ter simplesmente escrito:

```
def cons():
    n = int(input())
    d = { }
    for i in range(n):
        k,v = input().split()
        if k in d: d[k] += int(v)
        else: d[k] = int(v)
    return d
def read_data():
    data = []
    n = int(input())
    for i in range(n):
        k,v = input().split()
        data.append((k, int(v)))
    return data

def consolidate(xs):
    d = { }
    print(consolidate(data))
    for (k, v) in xs:
        if k in d: d[k] += v
        else: d[k] = v
    return d

def main():
    data = read_data()
    print(consolidate(data))
    print(consolidate(data))
```

Vamos começar com um exemplo tradicional. Lembre-se que o fatorial de um número inteiro n >= 0 é o número n * (n-1) * ...* 2 * 1. Assim, por exemplo, 5! = 5*4*3*2*1. Se você observar com cuidado a expressão n*(n-1)*...*2*1 notará que, embora óbvia, ela é escorregadia. Afinal, há um significado implícito nos '...' e é necessário contar como bom senso do leitor para decidir o significado dos '...'. Vamos exibir uma definição mais precisa. Para isso vamos definir a expressão n!, o fatorial de n:

(1) 0! = 1

n! = n * (n-1)! para cada inteiro n >= 1.

Eis como 5! pode ser desdobrado de acordo com esta definição:

5! = 5 * 4!

= 5 * 4 * 3!

= 5 * 4 * 3 * 2!

= 5 * 4 * 3 * 2 * 1!

= 5 * 4 * 3 * 2 * 1 * 0!

= 5 * 4 * 3 * 2 * 1 * 1

Uma definição como (1) é dita recursiva. Ela possui dois ingredientes. A primeira linha de definição

$0! = 1$

nos mostra como obter diretamente o valor do fatorial de um número; um caso como este é chamado de base da definição recursiva. Por outro lado, a segunda linha

$n! = n * (n-1)!$

nos mostra como obter o fatorial de um número n usando o fatorial de um número menor; um caso como este é chamado de passo da definição recursiva.

Eis uma implementação em Python da definição recursiva do fatorial de um inteiro não-negativo n.

```
def fat(n):
    if n == 0: return 1
    return fat(n-1) * n
```

Considere agora o seguinte problema: queremos escrever uma função, digamos lrng(n), que recebe um inteiro n >= 0 e devolve uma lista com os números de 1 até n. Por exemplo ,a chamada lrng(5) deve devolver a lista [1, 2, 3, 4, 5] e a chamada lrng(0) deve devolver a lista []. Considere a seguinte definição recursiva:

(2) lrng(0) = []

lrng(n) = lrng(n-1) + [n], para cada inteiro n >= 1.

Assim,

lrng(5) = lrng(4) + [5]
= (lrng(3) + [4]) + [5]]
= ((lrng(2) + [3]) + [4]) + [5]
= (((lrng(1) + [2]) + [3]) + [4]) + [5]
= ((((lrng(0) + [1]) + [2]) + [3]) + [4]) + [5]
= (((([] + [1]) + [2]) + [3]) + [4]) + [5]
= [1, 2, 3, 4, 5]

A seguinte função em Python recebe um inteiro n >= 0 e produz uma lista com os números de 1 até n:

```
def lrng(n):
    if n == 0: return []
    return lrng(n-1) + [n]
```

Note que poderíamos alternativamente definir:

(3) lrng2(0) = []
lrng2(1) = [1]
lrng2(n) = lrng2(n-1) + [n], para cada inteiro n >= 2

Assim,

lrng2(5) = lrng2(4) + [5]
= (lrng2(3) + [4]) + [5]
= ((lrng2(2) + [3]) + [4]) + [5]
= (((lrng2(1) + [2]) + [3]) + [4]) + [5]
= ((([1] + [2]) + [3]) + [4]) + [5]
= [1, 2, 3, 4, 5]

Não é difícil se convencer de que lrng(n) = lrng2(n) para cada n >= 0. Este exemplo ilustra que a base de uma definição recursiva pode ser constituída de mais de um caso.

Vamos a um outro exemplo. Suponha, agora, que queremos escrever uma função que recebe um inteiro n >= 0 e devolve uma lista com os números de n até 1. Eis tal função

```
def gnrl(n):
    if n == 0: return []
    return [n] + gnrl(n-1)
```

Vamos pausar por um momento e destacar como você deve pensar a respeito de tais funções. Primeiro, você deve observar se a sua função devolve o resultado correto na base da recursão. No exemplo, a base é quando n é igual a 0, e a função devolve a lista vazia, como é de se esperar .Agora, você deve SUPOR que a sua função faz o que promete desde que submetida à instância menor do mesmo problema.

Assim, você deve supor que a chamada a gnrl(n-1), quando n > 0, e já que n-1 < n, produzirá a lista [n-1, n-2, ..., 2, 1]. Agora você deve mostrar como, a partir da solução para a instância menor, pode-se obter uma solução para o problema original.

Desta forma, se gnrl(n-1) devolve a lista [n-1, n-2, ..., 2, 1], então gnrl(n) devolve a lista [n] + [n-1, n-2, ..., 2, 1] = [n, n-1, ..., 1], que é a lista esperada.

Vamos resumir o que vimos até agora. Queremos resolver um problema P(n) para todo inteiro n (chamado de tamanho do problema) maior ou igual a um certo inteiro k (nos exemplos anteriores, k = 0). Para isso, você pode usar a seguinte estratégia:
(a) Mostre como resolver diretamente o problema P(k).

(b) Suponha que n > k e que você sabe resolver qualquer problema P(m) tal que k <= m < n. Mostre como resolver P(n) com esta informação.

Suponha que queremos encontrar a soma dos elementos de uma lista xs de números. O tamanho do problema que queremos resolver é o comprimento da lista xs. Se len(xs) = 0, então a soma dos elementos de xs é igual a 0.Suponha que len(xs) > 0. Agora, você deve pensar em um problema menor e de mesma natureza, digamos, calcular a soma dos elementos de xs[:-1]. Suponha que você tem a resposta s para esse problema. Então s + xs[-1] é soma dos elementos de xs. Isso dá origem ao seguinte algoritmo recursivo:

```
def sum1(xs):
    if len(xs) == 0: return 0
    return sum1(xs[:-1]) + xs[-1]
```

Considere agora a seguinte versão:

```
def sum2(xs):
    if len(xs) == 0: return 0
    return sum2(xs[:-2]) + xs[-2] + xs[-1]
```

Executando as duas versões:

>>> sum1([1,2,3,4,5])
15
>>> sum2([1,2,3,4,5])
Traceback (most recent call last): File "<pyshell>", line 1, in <module> File "/home/linked/rec.py", line 29, in sum2 return sum2(xs[:-2]) + xs[-2] + xs[-1] File "/home/linked/rec.py", line 29, in sum2 return sum2(xs[:-2]) + xs[-2] + xs[-1] File "/home/linked/rec.py", line 29, in sum2 return sum2(xs[:-2]) + xs[-2] + xs[-1]IndexError: list index out of range

Você consegue determinar onde está o erro? Note que o comando return sum2(xs[:-2]) + xs[-2] + xs[-1] será executado desde que len(xs) > 0. Em particular, ele será executado quando len(xs) = 1.No entanto, se len(xs) = 1, então -2 não é um índice válido para xs. Uma forma correta de implementar a ideia de sum2 é:

```
def sum2(xs):
    if len(xs) == 0: return 0
    return sum2(xs[:-2]) + xs[-2] + xs[-1]
```

Para finalizar esta introdução aos algoritmos recursivos, vamos ilustrar uma técnica chamada de divisão e conquista. Suponha que queremos resolver o mesmo problema, i.e., somar os elementos de uma lista. Se a lista não tem elementos, a soma vale 0. Se a lista tem um único elemento, a soma vale este único elemento. Suponha que a lista tem ao menos dois elementos. Então a soma dos elementos da lista pode ser realizada da seguinte forma:
(1) somamos primeiro os elementos da primeira metade da lista, obtendo s1;
(2) somamos agora os elementos da segunda metade da lista, obtendo s2;
(3) a somados elementos da lista é então s1 + s2.

```
def sumdc(xs):
    if not xs: return 0
    if len(xs) == 1: return xs[0]
    m = len(xs)//2
    return sumdc(xs[:m]) + sumdc(xs[m:])
```

Módulo 08

COMECE A PROGRAMAR COM PYTHON

Sobre o Módulo 7

Revisão

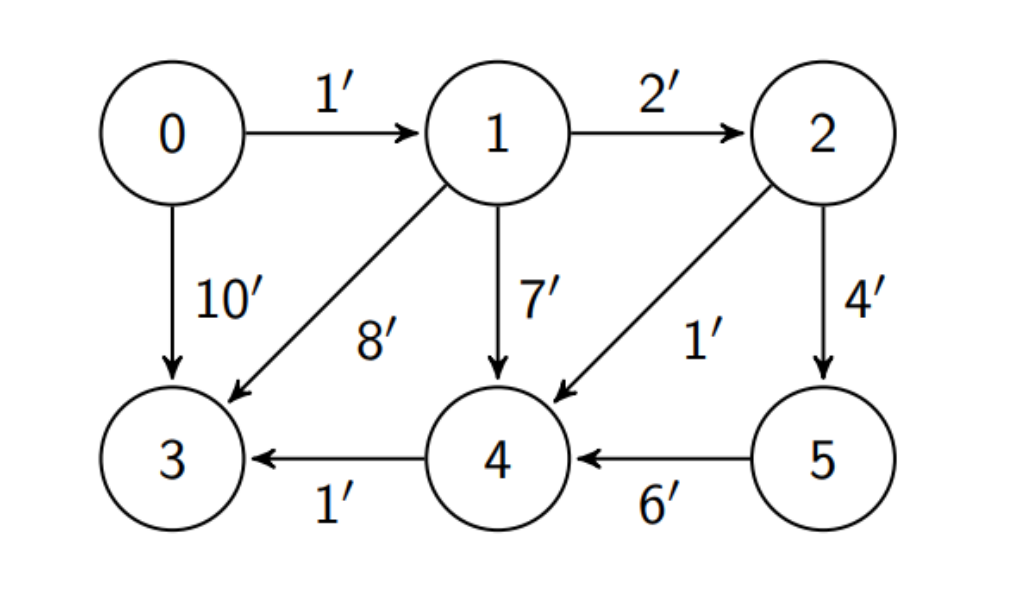
- ✓ Dicionários
- ✓ Recursão

Preliminares

Você está acostumado a se locomover usando um GPS ou o Waze.

- Vamos mostrar como se faz isso aproximadamente.
- O importante é minimizar a quantidade de tempo do trajeto escolhido.
- Assim, podemos ignorar o formato das ruas já que o importante é o tempo.
- Vamos admitir que são dados o mapa e o tempo que leva para atravessar cada rua do mapa.
- O mapa deve ser pensado como um objeto matemático que chamamos de digrafo.

Digrafos



No desenho há dois tipos de objetos: vértices e arcos.

• Você pode pensar nos vértices como representando os cruzamentos das ruas, e os arcos, representando as ruas.

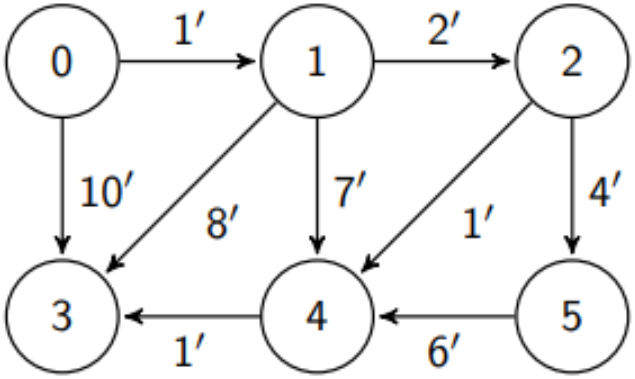
• No desenho, os números que estão acima dos arcos representam o tempo levado para atravessar o arco (ou a rua).

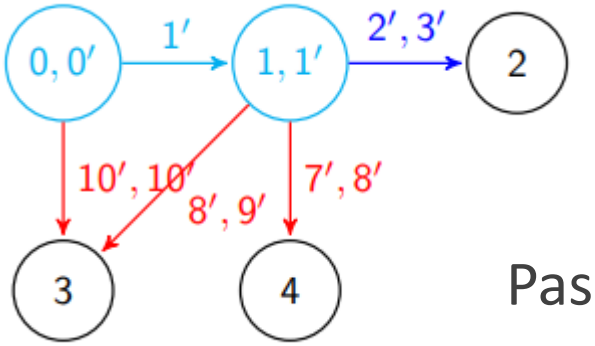
Digrafos

Seja X o conjunto dos vértices para o qual já calculamos um caminho mínimo, cuja cor será azul claro.

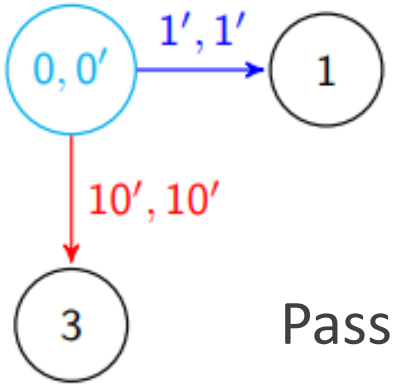
- A cor vermelha é o leque de saída de X, ou seja as ruas que permitem sair de X
- Nas rua do leque colocamos dois números: o primeiro é o tempo que leva para atravessar a rua, e o segundo, é o tempo total para chegar até onde a rua leva.
- E finalmente, de azul escuro colocamos a rua selecionada, que é aquela que permite a forma mais rápida de sair de X.

Caminhos Mínimos

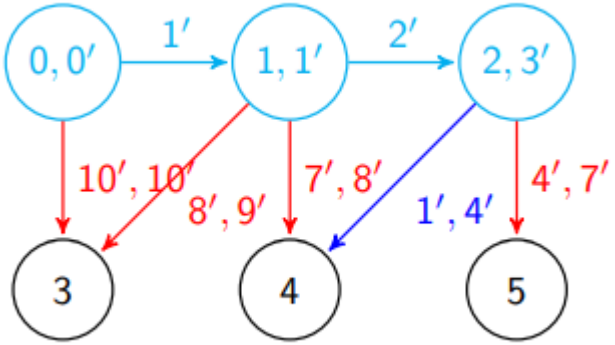




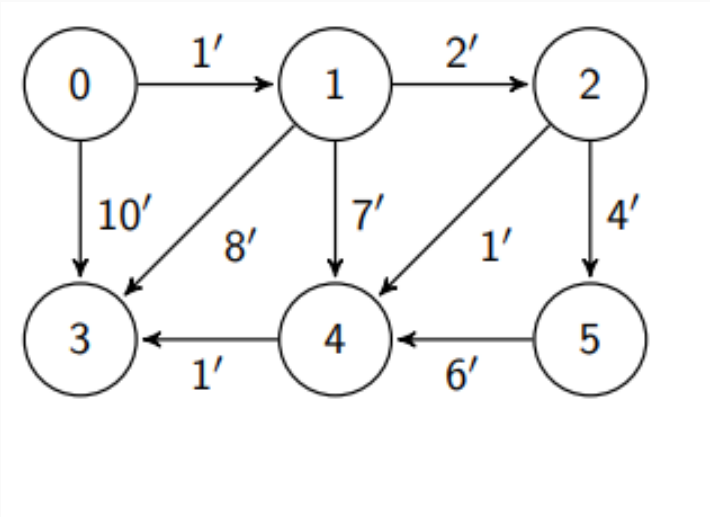
Passo 2

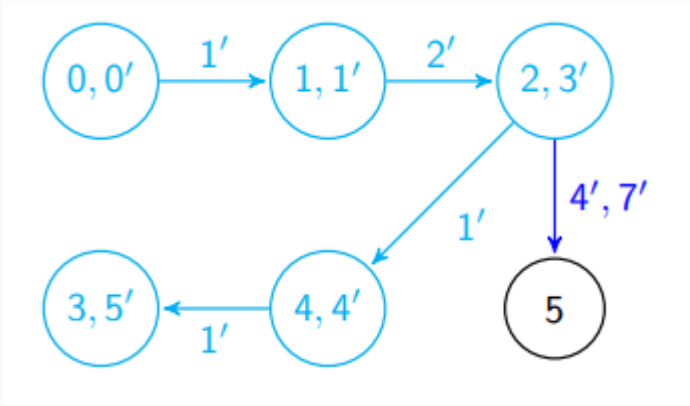


Passo 1

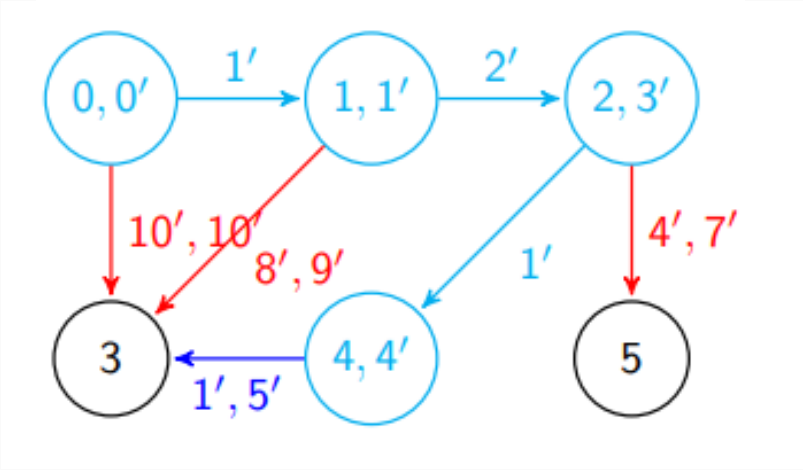


Passo 3

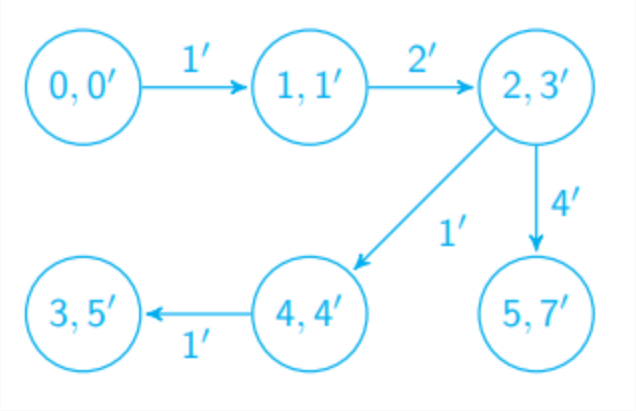




Passo 5



Passo 4



Passo 6

Possível implementação em Python (parte 1)

```
def min_path(G, s):
    dist = [None for i in range(len(G))]
    dist[s] = 0
    pred = [None for i in range(len(G))]
    pred[s] = s
    while True:
        leq = [(x, c, y) for x in range(len(G)) if dist[x] != None
        for (c, y) in G[x] if dist[y] == None]
        if leq == []: break
        (x, c, y) = min(leq, key = lambda t: dist[t[0]] + t[1])
        dist[y] = dist[x] + c
        pred[y] = x
    return (pred, dist)

def get_path(pred, t):
    if t == pred[t]: return str(t)
    return get_path(pred, pred[t]) + ' ' + str(t)
```

Caminhos Mínimos

Possível implementação em Python (parte 2)

```
def main():
    [v, a] = [int(k) for k in input().split()]
    G = [[] for i in range(v)]
    for i in range(a):
        [x, c, y] = [int(k) for k in input().split()]
        G[x].append((c, y))
    s = int(input())
    (pred, dist) = min_path(G, s)
    for t in range(len(G)):
        print('dist(%d, %d) = %d' %(s, t, dist[t]))
        print ('caminho: %s' %(get_path(pred, t)))
        print('')
```

Cursos com Alta Performance de Aprendizado