Inteligência Artificial

Aula 7

Profª Bianca Zadrozny

http://www.ic.uff.br/~bianca/ia

Linguagem Python

- Introdução
  - Características de Python
  - Rodando programas
  - Módulos
- Tipos básicos
  - Números e variáveis
  - Strings
  - Listas e tuplas
  - Dicionários
- Fluxo de controle
  - Execução condicional
  - Repetições

Características de Python

- Gratuita. Roda em muitas plataformas.
  - Pode ser baixada em www.python.org
- Fácil de ler.
  - Ao contrário de Perl = "write only language"
- Tempo de implementação rápido.
  - Ao contrário de Java.
- Orientada a objeto.

Usando Python no modo interativo

- Usuário digitando em vermelho, máquina respondendo em preto.

```
$ python
>>> print "Hello everyone!"
Hello everyone!
>>> print 2+2
4
>>> myname = "Andrew"
>>> myname
'Andrew'
```


Módulos

- Arquivos texto contendo código Python.

foo.py

```
print 25*3                          # multiply by 3
print 'CompLing ' + 'lecture 3'     # concatenate with +
myname = 'Andrew'
```

Módulo sendo executado a partir da linha de comando:

```
$ python foo.py
75
CompLing lecture 3
$
```


Importando módulos

- Todo arquivo terminando em **`.py`** é um módulo python que pode ser importado.
  - A terminação **`.py`** é omitida.
  - As declarações do módulo (funções, variáveis) ficam disponíveis como atributos de um objeto que tem o nome do módulo.

```
$ python
>>> import foo
75
CompLing lecture 3
>>> foo.myname
'Andrew'
```

Recarregando módulos

- A importação de módulos é cara computacionalmente e Python só a realiza uma vez (mesmo que o arquivo seja editado).
- Para forçar que o arquivo seja importado novamente, devemos usar o comando "reload".

Edit foo.py to print 25*4 (instead of 25*3) and reload

```
>>> reload(foo)
100
CompLing lecture 3
<module 'foo' from 'foo.py'>
```


Atributos de Módulos

- Considere o arquivo **`bar.py`**

```
university = 'UMass'
department = 'Linguistics'
```

```
>>> import bar
>>> print bar.department
Linguistics

>>> from bar import department
>>> print department
Linguistics

>>> from bar import *
>>> print university
UMass
```


Estruturas de Programas em Python

- Programas são compostos de módulos.
- Módulos contém comandos.
- Comandos contém expressões.
- Expressões criam e processam objetos.

- Comandos incluem:
  - Atribuição de variáveis
  - Chamadas de função
  - Controle de fluxo
  - Declaração de função
  - Declaração de objeto
  - Leitura e impressão

Objetos "built-in" de Python

- Números: inteiro e ponto-flutuante
- Strings
- Listas
- Dicionários
- Tuplas
- Arquivos

Expressões numéricas e variáveis

- Operadores usuais: +, *, /, **
- Precedência usual:
  - A * B + C * D = (A*B) + (C*D)
- Módulos úteis: **`math`** e **`random`**

- Variáveis
  - Criadas na primeira atribuição
  - São substituídas pelo seu valor quando usadas em expressões
  - Tem que ter recebido um valor antes do primeiro uso.
  - Não precisam ser declaradas

Strings

- Manipular strings em Python é fácil.
  - Comparado com linguagens como C, Java e C++.
- Strings podem ser escritas usando aspas simples ou aspas duplas.
  - Torna mais fácil incluir aspas dentro da string.
  - Exemplos:
    - **'Isso é uma string em Python'**
    - **"Isso também é uma string em Python"**
    - **'Ele disse "Alô?" e desligou o telefone'**
    - **"Gota d'água"**

Barra invertida em strings

- A barra invertida pode ser usada como caracter de escape para caracteres especiais.
- Exemplo: \n é newline, \t é tab

```
>>> s = 'Name\tAge\nJohn\t21\nBob\t44'
>>> print s
Name        Age
John        21
Bob         44
>>> t = '"Mary\'s"'
>>> print t
"Mary's"
```


Aspas triplas

- Aspas triplas (""" ou ''') são usadas para strings que ocupam várias linhas.

```
>>> s = """this is
... a string
... over 3 lines"""
>>> t = '''so
... is
... this'''
>>> print s
this is
a string
over 3 lines
>>> print t
so
is
this
```


Operações com Strings

- Concatenação (+)
- Comprimento (len)
- Repetição (*)
- Indexação e "Slicing" ([ ])

```
s = 'computational'
t = 'linguistics'
cl = s + ' ' + t        # 'computational linguistics'
l = len(cl)             # 25
u = '-' * 6             # ------
c = s[3]                # p
x = cl[11:16]           # 'al li'
y = cl[20:]             # 'stics'
z = cl[:-1]             # 'computational linguistic'
```


Indexação e Quebra

Figure 7-1. Offsets and slices: positive offsets start from the left end (offset 0 is the first item), and negatives count back from the right end (offset –1 is the last item). Either kind of offset can be used to give positions in indexing and slicing.

Métodos de strings

- Alguns exemplos:

```
s = 'example'
s = s.capitalize()          # 'Example'
t = s.lower()               # 'example'
flag = s.isalpha()          # True
s = s.replace('amp','M')    # 'exMle'
i = t.find('xa')            # 1
n = t.count('e')            # 2
```


Listas em Python

- Listas são coleções ordenadas de objetos arbitrários.
- Objetos são acessados usando índices que indicam a posição na lista a partir do início.
- Têm tamanho variável (crescem automaticamente à medida que os objetos são inseridos).
- São heterogêneas, isto é, podem conter qualquer tipo de objeto, incluindo outras listas.
- São mutáveis, ao contrário de strings.

```
>>> s = ['a', 'b', 'c']
>>> t = [1, 2, 3]
>>> u = s + t              # ['a', 'b', 'c', 1, 2, 3]
>>> n = len(u)             # 6
```

Indexação e “Slicing” de Listas

- Funciona da mesma forma que com strings.
- A indexação retorna o objeto em uma dada posição.
- O “slicing” retorna uma lista.
- Podemos usar indexação e “slicing” para mudar o conteúdo da lista.

```
l = [‘a’, ‘b’, ‘c’, ‘d’]
x = l[2]                    # ‘c’
m = l[1:]                   # [‘b’, ‘c’, ‘d’]
l[2] = ‘z’                  # [‘a’, ‘b’, ‘z’, ‘d’]
l[0:2] = [‘x’, ‘y’]         # [‘x’, ‘y’, ‘z’, ‘d’]
```


Métodos de listas

- Alguns exemplos:

```
l = [7, 8, 9, 3]
l.sort ()               # [3, 7, 8, 9]
l.append(6)             # [3, 7, 8, 9, 6]
l.append([1, 2])        # [3, 7, 8, 9, [1, 2]]
l.extend([‘r’, ‘s’])    # [3, 7, 8, 9, [1, 2],‘r’, ‘s’]
```


Dicionários

- Dicionários são endereçados por *chave*, não por posição.
  - Podem ser vistos como uma coleção de pares chave:valor.
- São coleções *não-ordenadas* de objetos arbitrários.
- Tem tamanho *variável* e podem conter objetos de qualquer tipo, inclusive outros dicionários.
- São *mutáveis* como as listas.

Exemplo com dicionários

```
level = {‘low’:1, ‘medium’:5}
x = level[‘medium’]           # 5
n = len(level)                # 2

flag = level.has_key(‘low’)   # True
l = level.keys()              # [‘low’,‘medium’]

level[‘low’] = 2      # {‘low’:2, ‘medium’:5}
level[‘high’] = 10    # {‘low’:2, ‘high’:10, ‘medium’:5}

level.items()
[(‘low’,2), (‘high’,10), (‘medium’,5)]

level.values()
[2, 10, 5]
```


Sobre dicionários

- Operações sobre seqüências não funcionam (ex.: “slice”) já que dicionários são mapeamentos e não seqüências.
- Dicionários têm um conjunto de chaves.
  - Só pode haver um valor por chave.
- Atribuir um valor a uma nova chave adiciona uma nova entrada ao dicionário.
- As chaves podem ser qualquer tipo de objeto.

- Dicionários podem ser usados como “records”.
- Dicionários podem ser usados para armazenar matrizes esparsas.

Outros Objetos

- Tuplas: como listas, porém imutáveis.

```
emptyT = ()
t1 = (1, 2, 3)
x = t1[1]          # 2
n = len(t1)        # 3
y = t1[1:]         # (2, 3)
```

- Arquivos: objetos com métodos para ler e escrever arquivos.

```
file = open(‘myfile’, ‘w’)
file.write(‘hellow file\n’)
file.close()

f2 = open(‘myfile’, ‘r’)
s = f2.readline()             # ‘hello file\n’
t = f2.readline()             # ‘’
all = open(‘myfile’).read()   #entire file as a string
```

Execução Condicional

```
course = 'Syntax'
if course == 'Syntax':
  print 'Bhatt'
  print 'or Potts'
elif course == 'Computational Linguistics':
  print 'McCallum'
else:
  print 'Someone else'
```

- A indentação determina a estrutura do bloco.
  - É o único lugar onde o espaço em branco importa.
- A indentação ajuda na legibilidade do código.
- Expressões depois do if e elif podem ser de quase qualquer tipo.
  - False, 0, [], (), '' funcionam como falso, o resto é verdadeiro.

Laços "While"

- Um laço do tipo "while" continua enquanto a expressão no topo for verdadeira.

```
a = 0
b = 10
while a < b:
  print a
  a = a + 1
```

```
s = 'abcdefg'
while len(s) > 0:
  print s
  s = s[1:]
```


Laços "For"

- "For" é usado pra percorrer uma seqüência qualquer de objetos.

```
l = ['a', 'b', 'c']
for i in l:
  print i

sum = 0
for x in [1, 2, 3, 4, 5, 6]:
  sum = sum + x
print sum
```

- O uso de "range" pode ser útil.

```
range(5)        # [0, 1, 2, 3, 4]
range(2,5)      # [2, 3, 4]
range(0,6,2)    # [0, 2, 4]
```


Laços "For"

- Fazer alguma coisa com cada item de uma lista.

```
l = [1, 2, 3, 4, 5, 6]  # or l = range(1,7)

# one way to print the square
for x in l:
  print x*x

# another way to do it
n = len(l)
for i in range(n):
  print l[i]*l[i]
```


Exemplo: Interseção

```
l1 = ['a', 'd', 'f', 'g']
l2 = ['a', 'b', 'c', 'd']
# one way
result = []
for x in l1:
  for y in l2:
      if x == y:
          result.append(x)
# or, alternatively
result = []
for x in l1:
  if x in l2:
      result.append(x)        # result == ['a', 'd']
```


Funções "built-in", importadas e definidas pelo usuário

- "Built-in"

```
l = len(['a', 'b', 'c'])
```

- Importadas

```
import math
from os import getcwd
print getcwd() # which directory am I in?
x = math.sqrt(9)    # 3
```

- Definidas pelo usuário

```
def multiply(a, b):
    return a * b
print multiply(4,5)
print multiply('-',5)
```

Definição de funções

- **`Def`** cria um objeto do tipo função e dá um nome a ele.
- **`Return`** retorna um objeto a quem chamou a função.

```
def square(x):       # create and assign
     return x*x
y = square(5)        # y gets 25
```


Exemplo

```
def intersect(seq1, seq2)
     result = []
     for x in seq1:
          if x in seq2:
               result.append(x)
     return result
```


Variáveis locais

- Variáveis dentro de uma função são locais àquela função.

```
>>> intersect(s1, s2):
...   result = []
...   for x in s1:
...        if x in s2:
...             result.append(x)
...   return result
...
>>> intersect([1,2,3,4], [1,5,6,4])
[1, 4]
>>> result
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'result' is not defined
```


Passagem de Parâmetros

- Objetos imutáveis são passados "por valor".

```
>>> def plusone(x):
...   x = x + 1
...   return x
...
>>> plusone(3)
4
>>> x = 6
>>> plusone(x)
7
>>> x
6
```


Passando parâmetros mutáveis

- Números, strings e tuplas são imutáveis enquanto listas e dicionários são mutáveis.
- Objetos mutáveis são passados "por referência".

```
>>> def appendone(s):
...   s.append('one')
...   return s
...
>>> appendone(['a', 'b'])
['a', 'b', 'one']
>>> l = ['x', 'y']
>>> appendone(l)
['x, 'y', 'one']
>>> l
['x', 'y', 'one']
```


map

```
>>> counters = range(1,6)
>>> updated = []
>>> for x in counters:
...   updated.append(x+3)
...
>>> updated
[4, 5, 6, 7, 8]

# Another way...
>>> def addthree(x):
...   return x+3
...
# map() applies a function to all elements of a list
>>> map(addthree, counters)
[4, 5, 6, 7, 8]
```

Funções anônimas

```
# lambda is a way to define a function with no name
>>> map((lambda x: x+3), counters)
[4, 5, 6, 7, 8]


# a list comprehension does something similar,
# but can offer more flexibility
>>> result = [addthree(x) for x in counters]
>>> result
[4, 5, 6, 7, 8]
>>> [addthree(x) for x in counters if x < 4]
[4, 5, 6]
```


Número variável de parâmetros

```
def max (*a):
  maximum = 9999999
  for x in a:
      if x > maximum:
          maximum = x
  return maximum
```


Parâmetros opcionais

```
def exp (x, exponent=2.718):
  return exponent ** x
```

```
>>> exp(1)
2.718
>>> exp(1, 2.0)
2.0
>>> exp(3, 2.0)
8.0
>>> exp(3, exponent=2.0)
8.0
```


Múltiplos parâmetros opcionais

```
def exp_plus (x, exponent=2.718, addend=0):
  return (exponent ** x) + addend
```

```
>>> exp(1)
2.718
>>> exp(1, 2.0)
2.0
>>> exp(1, exponent=2.0)
2.0
>>> exp(1, addend=2.0)
4.718
```


Número arbitrário de parâmetros opcionais

- A notação ** recebe todos os parâmetros extra em um dicionário.

```
def showargs (separator, **d):
  for key in d.keys():
      print str(key)+":"+str(d[key])+separator,
  print
```

```
>>> showargs(";", bi=2, tri=3, quad=4)
tri:3;bi:2;quad:4;
```

```
def showargs (separator, **d):
  for (key,val) in d.items():
      print str(key)+":"+str(val)+separator,
  print
```