Statistical Natural Language Processing

Python Introduction

Michael Watkins & Marko Lozajic[1]

April 20, 2018

[1] Based *heavily* on slides by Verena Blaschke and Kuan Yu.

Outline

Installation

Arithmetics

Data Structures
- Sequences: List, Tuple, Range, Str
- Set, Frozenset, Dict

Control Structures

Functions
- More: File I/O, Statements, ...

Installation

Check if Python 3.x.x (!) is already installed: `python3 -V` (or `python -V`)

If it is not:
Ubuntu:

```
sudo apt-get update
sudo apt-get install python3
```

Homebrew (Mac OS):

```
brew update
brew install python3
```

otherwise:
https://www.python.org/downloads/release/python-365/
https://www.anaconda.com/download/[2]

[2] It is difficult to install certain useful Python libraries (like SciPy) on Windows. Luckily, SciPy is already included in Anaconda.

Packages

package management via `pip`[‡]

```
pip install --upgrade pip
```

`https://packaging.python.org/tutorials/installing-packages/`

NumPy

```
pip install --upgrade numpy
```

IPython

```
pip install --upgrade ipython
Ubuntu: sudo apt-get install ipython
Mac?: brew install ipython
```

[‡] might be pip3 on your machine

Running Python code

Read-eval-print loop (REPL):

```
python3

>>> help()

>>> quit()
# or CTRL-D (Unix)
# or CTRL-C/CTRL-Z (Windows)
```

Run code from files:

```
python3 path/to/my/script.py
```

IDEs (PyCharm, ...)

Arithmetics

```
>>> 1 == 1
True
>>> 1 == 1.0
True
>>> 1 == int("1")
True
>>> 1 == float("1")
True
>>> # bool is a subclass of int
... 1 == True
True
>>> 0 == False
True
```

Arithmetics

```
>>> (1 + 2) * 3
9
>>> 4 ** 2
16
>>> 4 ** 0.5
2.0
>>> # float division is the default
... 5 / 3
1.6666666666666667
>>> 5 // 3
1
```

Sequences

```
>>> mylist = ['a', 'b', 3, 'd', 'e']
>>> mylist[0] # indexing
'a'
>>> mylist[4] == mylist[-1] == 'e'
True
>>> mylist[:2] # slicing
['a', 'b']
>>> mylist[-2:]
['d', 'e']
>>> mylist[1:4]
['b', 3, 'd']
>>> mylist[::2] # steps
['a', 3, 'e']
>>> mylist[::-1]
['e', 'd', 3, 'b', 'a']
```

Sequences

```
>>> mylist = ['a', 'b', 3, 'd', 'e']
>>> mylist + ['f', 'g']
['a', 'b', 3, 'd', 'e', 'f', 'g']
>>> mylist * 2
['a', 'b', 3, 'd', 'e', 'a', 'b', 3, 'd', 'e']
>>> 'b' in mylist
True
>>> 'c' not in mylist
True
>>> otherlist = [1, 1, 3, 3, 2, 4, 1]
>>> otherlist.count(3)
2
>>> otherlist.index(1) # first occurrence
0
>>> otherlist.index(1, 2)  # first occurrence at/after index 2
6
>>> len(otherlist)
7
```

Sequences

```
>>> otherlist = [1, 1, 3, 3, 2, 4, 1]
>>> # for sequences of numbers
... sum(otherlist)
15
>>> min(otherlist)
1
>>> max(otherlist)
4
```

```
>>> nestedlist = [['a', 'b', 'c'], [1, 2, 3], 4]
>>> len(nestedlist)
3
>>> nestedlist[0]
['a', 'b', 'c']
>>> nestedlist[0][1]
'b'
```

Sequences

```
>>> l = ['a', 'b', 'c', 'd', 'e']
>>> for idx, elem in enumerate(l):
...     '{} at index {}'.format(elem, idx)
...
'a at index 0'
'b at index 1'
'c at index 2'
'd at index 3'
'e at index 4'
>>> {elem: idx for idx, elem in enumerate(l)}
{'b': 1, 'e': 4, 'c': 2, 'd': 3, 'a': 0}
>>> l2 = ['v', 'w', 'x', 'y', 'z']
>>> {elem_l: elem_l2 for elem_l, elem_l2 in zip(l, l2)}
{'b': 'w', 'e': 'z', 'c': 'x', 'd': 'y', 'a': 'v'}
```

List

List: a **mutable** array

```
>>> mylist = ['a', 'b', 3, 'd', 'e']
>>> mylist[2] = 'c'
>>> mylist
['a', 'b', 'c', 'd', 'e']
>>> mylist += ['f', 'g']
>>> mylist.extend(['h', 'i'])
>>> mylist.append('j')
>>> mylist
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
>>> mylist.remove('c')
>>> mylist
['a', 'b', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
>>> mylist.insert(2, 'c')
>>> mylist
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

List

List: a **mutable** array

```
>>> mylist = ['a', 'b', 3, 'd', 'e']
>>> elem = mylist.pop(2)
>>> "elem: {}, mylist: {}".format(elem, mylist)
"elem: 3, mylist: ['a', 'b', 'd', 'e']"
>>> elem = mylist.pop()
>>> "elem: {}, mylist: {}".format(elem, mylist)
"elem: e, mylist: ['a', 'b', 'd']"
>>> mylist.reverse()
>>> mylist
['d', 'b', 'a']
>>> mylist.sort()
>>> mylist
['a', 'b', 'd']
>>> mylist.clear()
>>> mylist
[]
```

Tuple

Tuple: an **immutable** array

- the general sequence operations also work for tuples

```
>>> mytuple = (1, 2, 3, 4)
>>> mytuple[-2:]
(3, 4)
>>> mytuple + (5, 6) # returns a new tuple
(1, 2, 3, 4, 5, 6)
>>> len(mytuple)
4
>>> 2 in mytuple
True
>>> # immutable!
... mytuple[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'tuple' object does not support item assignment
```

Range

Range: an **immutable** sequence of numbers

- the general sequence operations also work for ranges
- takes a small amount of memory that does not increase with the length of the sequence

```
>>> myrange = range(10) # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
>>> myrange[8]
8
>>> myrange[:-3]
range(0, 7)
>>> 7 in myrange
True
>>> tuple(myrange)
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> list(range(1, 9)) # creating ranges is similar to slicing
[1, 2, 3, 4, 5, 6, 7, 8]
>>> list(range(1, 10, 2))
[1, 3, 5, 7, 9]
```

String

String: an **immutable** sequence of Unicode code points

- the general sequence operations also work for strings

```
>>> s = 'linguistics' # double or triple quotes work too
>>> long_string = 'this is a very, very, very, very, ' \
...               'very, very, very, very long string'
>>> s[-2]
'c'
>>> s[-2:]
'cs'
>>> s.count('i')
3
>>> 'g' in s
True
```

String

```
>>> 'g' in s
True
>>> 'ling' in s # subsequence testing
True
>>> 'computational {}!'.format(s)
'computational linguistics!'
>>> t = "   I'm surrounded by whitespace  \n"
>>> t.strip()
"I'm surrounded by whitespace"
>>> t.split() # default delimiter: whitespace
["I'm", 'surrounded', 'by', 'whitespace']
```

Set

Set: a **mutable**, unordered collection of hashable objects

- mutable containers (sets, dictionaries) are **not** hashable

```
>>> myset = {3, 2, 'a', 'b'}
>>> # collection operations
... 5 in myset
False
>>> len(myset)
4
>>> # set-theoretic operations
... mysubset = {2, 3}
>>> mysubset.issubset(myset)
True
>>> myset.intersection(mysubset)
{2, 3}
```

Frozenset

Frozenset: an **immutable** set

```
>>> myset = {3, 2, 'a', 'b'}
>>> myfrozenset = frozenset(['a', 'b', 2, 3])
>>> myset == myfrozenset # compares members
True
```

Dictionary

Dictionary: a **mutable** mapping from hashable objects to arbitrary objects

```
>>> mydict = {'a': 1, 'b': 2, 'c': 3}
>>> mydict['a']
1
>>> mydict.items()
dict_items([('b', 2), ('c', 3), ('a', 1)])
>>> mydict.keys()
dict_keys(['b', 'c', 'a'])
>>> mydict.values()
dict_values([2, 3, 1])
>>> # can be used as a switch
... switchdict = {'sum': sum, 'len': len, 'min': min}
>>> switchdict['len'](mydict)
3
```

Comparisons

```
>>> 1 < 2 < 3
True
>>> True or 1/0 # lazy! no ZeroDivisionError
True
>>> not False and not None and not 0 \
... and not '' and not () and not [] and not {}
True
>>> l = [1, 2, 3]
>>> m = [1, 2, 3]
>>> # equality
... l == m
True
>>> # identity
>>> l is m
False
>>> l is not m
True
```

`if`, `elif`, `else`

`not`, `and`, `or`

Loops and List Comprehension

```
>>> squares = []
>>> for n in range(10):
...     squares += [n * n]
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> n # loops don't create a lexical scope!
9
>>>
>>> # as a one-liner (list comprehension)
... squares = [n * n for n in range(10)]
>>>
>>> some_squares = [x for x in squares if x % 3 == 0]
>>> some_squares
[0, 9, 36, 81]
```

Loops

```
>>> my_set = {'a', 'b', 'c', 'd', 'e'}
>>> while True:
...     elem = my_set.pop() # removes a random member
...     if elem == 'c':
...         break
...     print(elem)
...
a
>>> my_set
{'e', 'b', 'd'}
```

Iterators

```
>>> my_set = {'a', 'b', 'c', 'd', 'e'}
>>> it = iter(my_set)
>>> while True:
...     try:
...         print(next(it))
...     except StopIteration:
...         break
...
a
c
e
b
d
```

Functions

```python
def count_vowels(s, vowels=('a', 'e', 'i', 'o', 'u')):
    counts = {}
    for vowel in vowels:
        counts[vowel] = s.count(vowel)
    return counts
    # or all in one line:
    # return {vowel: s.count(vowel) for vowel in vowels}

print(count_vowels('linguistics'))
# {'u': 1, 'a': 0, 'o': 0, 'e': 0, 'i': 3}
```

built-in functions:

https://docs.python.org/3/library/functions.html

Functions

```python
def my_function(arg1, arg2='default value', *args, **kwargs):
    print('obligatory:', arg1)
    print('optional:', arg2)
    print('optional (positional):', args)
    print('optional (with keyword):', kwargs)

my_function(0, 1, 2, 3, 4, five=5, six=6)
# obligatory: 0
# optional: 1
# optional (positional): (2, 3, 4)
# optional (with keyword): {'five': 5, 'six': 6}
my_function(0)
# obligatory: 0
# optional: default value
# optional (positional): ()
# optional (with keyword): {}
```

Functions

```python
def my_function(arg1, arg2='default value', *args, **kwargs):
    print('obligatory:', arg1)
    print('optional:', arg2)
    print('optional (positional):', args)
    print('optional (with keyword):', kwargs)

my_list = [3, 4, 5]
my_function(0, *my_list) # unpacks the list
# obligatory: 0
# optional: 3
# optional (positional): (4, 5)
# optional (with keyword): {}
```

Functions

Avoid using mutable default arguments:

```
>>> def add(x, l=[]):
...     l.append(x)
...     return l
...
>>> add(1)
[1]
>>> add(2)
[1, 2]
>>> def add(x, l=None):
...     if l is None:
...         l = []
...     l.append(x)
...     return l
...
>>> add(1); add(2)
[1]
[2]
```

Functions

Functions can return multiple objects:

```
>>> def square_and_cube(x):
...     return x**2, x**3
...
>>> n = 3
>>> s, c = square_and_cube(n)
>>> 'n: {}, s: {}, c: {}'.format(n, s, c)
'n: 3, s: 9, c: 27'
```

File I/O

```python
f = open('file.txt', 'w', encoding='utf8')
f.write('Hi!')
f.close()

# Using `with` closes the input stream automatically,
# even if an exception is raised!
with open('file.txt', 'w', encoding='utf8') as f:
    f.write('Hi!')

with open('file2.txt', 'r', encoding='utf8') as f:
    for line in f:
        print(line + '!')

with open('file2') as f:
    list_of_lines = f.readlines()
```

https://docs.python.org/3/tutorial/inputoutput.html

Statements

```python
import my_module
import my_module as mm
from my_module import my_identifier

my_list = ['a', 'b', 'c', 'd', 'e']

# `assert` is useful for debugging
assert len(my_list) == 5

# Use `pass` when you need a syntactic placeholder:
def implement_me():
    pass # nothing happens

del my_list
# Referring to `my_list` now raises a NameError
```

`break`, `continue`

`return`, `yield`

`try`, `except`, `finally`, `raise`

Links

- Installing Python and Pip
  https://www.python.org/downloads/release/python-365/
  https://www.anaconda.com/download/
  https://packaging.python.org/tutorials/installing-packages/
- IDEs for Python
  https://wiki.python.org/moin/IntegratedDevelopmentEnvironments
- Python 3 Documentation
  https://docs.python.org/3/library/index.html
  https://docs.python.org/3/reference/index.html
  https://docs.python.org/3/tutorial/index.html
- PEP 8: Style Guide for Python
  https://www.python.org/dev/peps/pep-0008/