Python: An Introduction

CSE 415: Introduction to Artificial Intelligence
University of Washington
Spring, 2017

Why Python?

Python supports features needed in AI programming.

Python is increasingly popular, and high-quality, free tools are readily available.

Python's syntax means its code tends to be more readable than code in other languages.

Python has some advantages over Lisp in string processing and user-interface construction.

With the Python 3.x generation, the language is cleaner and more efficient.

Nice Things About Python for AI

- Supports rapid prototyping
  - Concise; minimal demands for declaration
  - Supports alternative ways of thinking; multiparadigm:
    - Imperative
    - Functional
    - Object-oriented
    - Scripting
  - Good tools are available: IDLE, PyCharm, PyDev, Emacs

More Nice Things

- Simple Syntax
  - Use of indentation to group program blocks
  - Readable but flexible
- Clean
  - Safer than Perl
- Free, open
- Cross-platform
  - Windows, OS/X, Linux
  - Jython: applets, Swing, and other Java libraries
- Rich Resources Available
  - Modules, often very efficient (coded in C).
  - Great community
- Can be run in web browsers: Brython, Skulpt.

Getting Started

- Download and install Python 3.6 and IDLE from www.python.org on your own computer.
- Optionally install Emacs, or Eclipse/PyDev or PyCharm from JetBrains.com.
- Read "Python as a Second Language."
- Browse some of the online Python resources, such as G. van Rossum's tutorial.
- Work on Assignment 1.

Data Types

```
• int       105
• float     3.52
• str       "text", 'apple'
• list      ['apple', 'banana', 'orange']
• bool      True, False
• tuple     ('apple', 'banana', 'orange')
• dict      {'one': 1, 'two': 2}
• function  lambda x: 2*x
• builtin_function_or_method  math.sqrt
```

Interacting with Python

```
>>> 5 + 7
12
>>> x = 3
>>> x * x
9
>>> s = 'apple'
>>> s+s
'appleapple'
>>> s2 = """Two line
string"""
```


Defining Functions

```
def sqr(x):
    return x*x

sqr(5)
25

sqr(25)
625

sqr(1.5)
2.25
```


Defining Recursive Functions

```
def fact(n):
    if n==1:
        return 1
    else:
        return n * fact(n-1)

fact(5)
120
```


Scopes of Bindings

```
x = 5
y = 6
z = 7
def foo(x):
    global y
    z = x + y
    return z

w = foo(4)
w
x
y
z
```


Lists

- Lists are sequences of elements separated by commas and surrounded by square brackets.
- `['a', 'b', 'c']`
- `[0, 1, 2]`
- `['testing', 1, 2, 3]`
- `[ ]`

List Elements

List elements may be extracted with an index in square brackets.

```
>>>  L = ['a', 'b', 'c']
>>>  L[0]
'a'
>>>  L[1]
'b'
>>>  L[2]
'c'
>>>  L[-1]
'c'
```

Lists are “zero-based”; indices start at 0. Negative indices work right-to-left.

Slices

Sublists are expressed using "slice" notation.

```
>>>  L2 = ['a', 'b', 'c', 'd', 'e']
>>>  L2[2:4]
['c', 'd']
>>>  L2[1:]
['b', 'c', 'd', 'e']
>>>  L2[:3]
['a', 'b', 'c']
>>>  L2[::2]
['a', 'c', 'e']
>>>  L2[:]     # copies L2.
```

Slices are copies of their original lists or sublists.

Strings

- Strings are sequences of characters surrounded by quotations marks (or by apostrophes).
- `"Hello"`
- `'Hello'`
- Multiline strings are delimited by pairs of triple quotes (or apostrophes).

```
"""Line 1
Line 2
Line 3
"""
```


String Elements

List elements may be extracted with an index in square brackets.

```
>>>  S = "abc"
>>>  S[0]
'a'
>>>  S[1]
'b'
>>> S[2]
'c'
>>>  S[-1]
'c'
```

Strings are "zero-based"; indices start at 0. Negative indices work right-to-left.

String Slices

Substrings are expressed using "slice" notation.

```
>>>  S2 = 'abcde'
>>>  S2[2:4]
'cd'
>>>  S2[1:]
'bcde'
>>>  S2[:3]
'abc'
>>>  S2[::2]
'ace'
>>>  S2[:]
```

Slices are copies of their original strings or substrings.

Dictionaries

Dictionaries are mappings from keys to values, implemented with hash tables.

```
>>> color = {}
>>> color['apple'] = 'red'
>>> color['kiwi'] = 'green'
>>> print(color['apple'])
'red'
```


KeyError with Dictionaries

```
color = {}
color['apple'] = 'red'
color['kiwi'] = 'green'
fruit = 'banana'

try:
  print(color[fruit])
except KeyError:
  print("There is a problem with "+fruit +"!")
```

There is a problem with banana!

Functional Programming

- Functions can be values, i.e., assigned to variables, elements of lists, etc.
- Functions can be arguments to other functions and returned by functions.
- Functions can be synthesized at run-time.
- Functions can be explicitly applied to arguments.
- Functions do not have to have names.
- Functions tend, but don't have to be, "pure," meaning without side effects and producing values that depend only upon the values of their parameters ("referentially transparent").

Anonymous Functions

First, here is a named function, named f:

```
>>>  f = lambda x: x+5
>>>  f(4)
9
>>>  list(map(f, [0, 1, 2, 3, 4]))
[5, 6, 7, 8, 9]
```

Second, we use an anonymous function:

```
>>>  list(map(lambda x: x*7, [0, 1, 2, 3, 4]))
[0, 7, 14, 21, 28]
```


Runtime Creation of Functions

```
>>>  def make_adder(y):
          return lambda x: x + y

>>>  f4 = make_adder(4)
>>>  f4(5)
9
>>>  f7 = make_adder(7)
>>>  f7(11)
18
```


Another Example

```
def make_quadratic_evaluator(a, b, c):
    return lambda x: a*x*x + b*x + c

fsqr = make_quadratic_evaluator(1,0,0)

print fsqr(5)
25
```

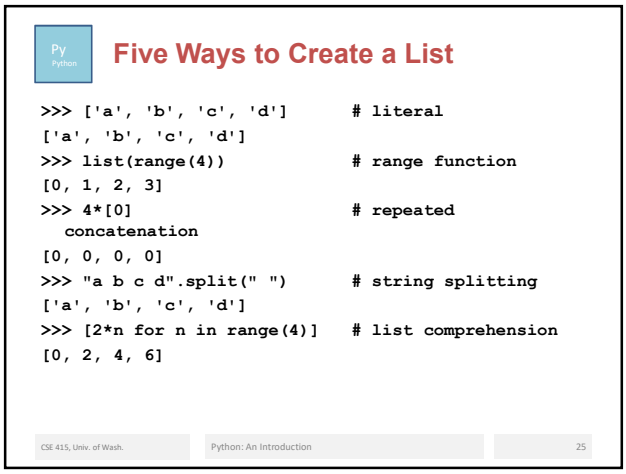

A Challenge…

Create a function that transforms other functions as follows.

```
>>> F1 = lambda n: n
>>> F2 = lambda n: n*n
>>> F3 = lambda n: n*n*n
>>> F1X = transform(F1)
>>> F1X(5)
11
>>> F2X = transform(F2)
>>> F2X(5)
61
>>> F3X = transform(F3)
>>> F3X(5)
341
```


Hint

If F transforms n into F(n), then
FX transforms n into F(n) + F(n+1)


Py
Python
Five Ways to Create a List
>>> ['a', 'b', 'c', 'd'] # literal
['a', 'b', 'c', 'd']
>>> list(range(4)) # range function
[0, 1, 2, 3]
>>> 4*[0] # repeated
concatenation
[0, 0, 0, 0]
>>> "a b c d".split(" ") # string splitting
['a', 'b', 'c', 'd']
>>> [2*n for n in range(4)] # list comprehension
[0, 2, 4, 6]
CSE 415, Univ. of Wash.
Python: An Introduction
25