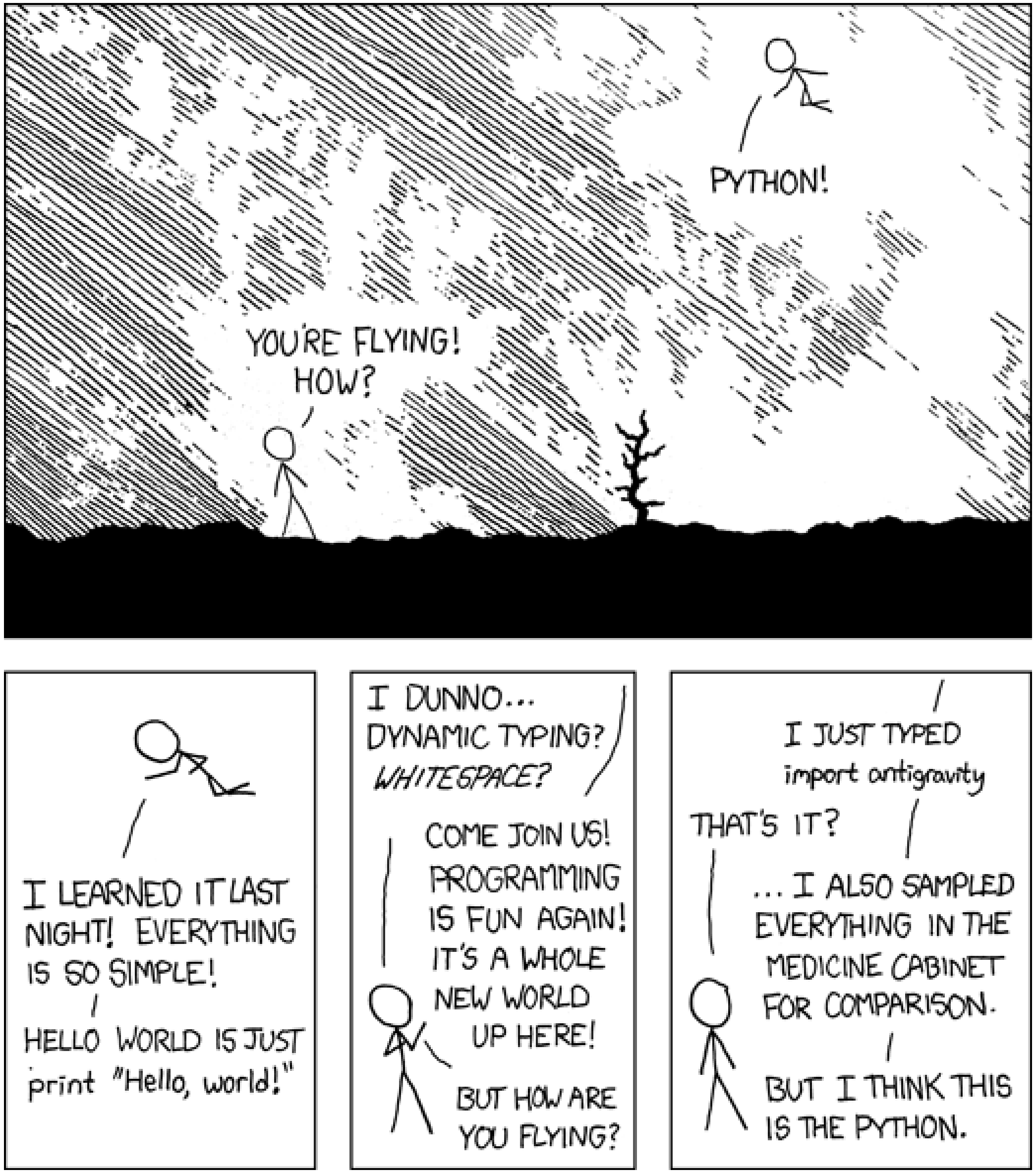
MY470 Computer Programming

Data Types in Python

Week 2 Lecture, MT 2017

Overview

- About Python
- Scalars: `int`, `float`, `bool`, `None`
- Operators: arithmetic, boolean, comparison, assignment, membership
- Non-scalars and methods for non-scalars
  - Unordered non-scalars: `set`, `dict`
  - Ordered non-scalars (sequences): `list`, `tuple`, `string`
- Mutability
  - Aliasing vs. cloning
  - List operations


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Source: http://xkcd.com/353/ (http://xkcd.com/353/)

Why Python?

- Open-source – free and well-documented
- Simple and concise syntax
- Many useful libraries
- Cross-platform
- Widely used in industry and science

Python vs. Java: Syntax

- Python

```
In [4]: print('Hello world!')
```

```
Hello world!
```

- Java

```
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Hello world!");
    }
}
```

Python vs. C, Matlab, R, and Julia: Speed

Task	Python	C	Matlab	R	Julia
Loops	61.97	0.55	6.80	744.93	0.34
Matrix multiplication	0.95	-	0.90	11.46	1.09
Open files and plot data	1399	-	1678	2220	1317
Metropolis-Hastings algorithm	0.08	4.30	0.99	28.63	0.73

Source: https://modelingguru.nasa.gov/docs/DOC-2625
(https://modelingguru.nasa.gov/docs/DOC-2625)

A Brief History of Python

- Started in December 1989 by Guido van Rossum, BDFL (Benevolent Dictator for Life)
- Python 2.0 released in 2000
- Python 3.0, which is backward-incompatible, released in 2008
- End of Life date for Python 2.7 has been postponed to 2020

From Last Week: Objects, Data Types, and Expressions

- Computer programs manipulate objects
- Objects have types
    - Scalar — indivisible
    - Non-scalar — with internal structure
- Expressions combine objects and operators

Scalar Data Types

- Integer
- Float
- Boolean
- NoneType
- (String is non-scalar in Python)

In [5]:

```
print(type(2))
print(type(1.125))
print(type(True))
print(type(None))
print(type('a'))
```

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'NoneType'>
<class 'str'>
```

Converting between Scalar Data Types

- Use the name of a type to convert values to that type

In [6]:

```
a = float(123)
b = int('32')
print(a, b)
```

```
123.0 32
```

Operators

- Arithmetic
- Boolean
- Comparison
- Assignment
- Membership

Arithmetic Operators

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` modulus
- `//` floor division
- `**` exponent

In [7]:

```
# + and * have different meanings depending on the types of objects with which the
y are used
print(2+2)
print('a'+'bc')
print(3*2)
print(3*'a'+'h!')
```

```
4
abc
6
aaah!
```

Boolean Operators

- `and`
- `or`
- `not`

In [8]:

```
print(True and False)
print(True or False)
print(not False)
```

```
False
True
True
```

Comparison Operators

- `==` equals
- `!=` does not equal
- `>` is greater than
- `<=` is less than or equal, etc.

Assignment Operators

- `=` assign right operand to left operand
- `+=` add right operand to left operand and assign to left operand
- `-=` subtract right operand from left operand and assign to left operand, etc.

In [2]:

```
a = 2
a +=3 # Equivalent to a = a + 3
print(a)
```

```
5
```

Comparison vs. Assignment Operators

In [9]:

```
a = 2 # This is assignment
print(a==1) # This is test for equality. It returns bool.
```

```
False
5
```

Membership Operators

- `in` left element is in right sequence

```
In [5]: print('x' not in 'abcdefg')
```

```
True
```

Non-Scalar Data Types

- List – a mutable ordered sequence of values
- Tuple – an immutable ordered sequence of values
- String – an immutable ordered sequence of characters
- Set – a mutable unordered collection of unique values
- Dictionary – a set of key/value pairs

In [7]:

```
list_var = [1, 2, 2, 'a', 'a'] # list
tuple_var = (1, 2, 'a', 'b') # tuple
set_var = {1 , 2, 2, 'a', 'b'} # set
dict_var = {1: 'a', 2: 'b', 3: 'c'} # dictionary
print(list_var, set_var)
```

```
[1, 2, 2, 'a', 'a'] {'b', 1, 2, 'a'}
```

Converting between Non-Scalar Data Types

- Use the name of a type to convert values to that type

In [12]:

```
tup = tuple([1, 2, 3])
dic = dict( [(1, 'a'), (2, 'b'), (3, 'c')] )
print(tup, dic)
```

```
(1, 2, 3) {1: 'a', 2: 'b', 3: 'c'}
```

Length of Non-Scalar Objects

- The `len()` function returns the length of the element

In [13]:

```
print( len( [0,1,2] ) )
print( len('ab') )
print( len( (1,2,3,4,'a') ) )
print( len( {1:'a', 2:'b'} ) )
```

```
3
2
5
2
```

Strings

- You can write string literals in different ways
  - Single quotes: `'allows embedded "double" quotes'`
  - Double quotes: `"allows embedded 'single' quotes"`
  - Triple quoted: `'''Three single quotes'''`, `"""Three double quotes"""`

```
In [14]: '''Triple quoted strings may span multiple lines -
         all associated whitespace will be included
         in the string literal.'''
```

```
Out[14]: 'Triple quoted strings may span multiple lines - \nall associated whitespace w
         ill be included \nin the string literal.'
```

- Strings implement all of the common sequence operations we will shortly discuss, along with some additional methods: http://docs.python.org/3/library/stdtypes.html#string-methods (http://docs.python.org/3/library/stdtypes.html#string-methods)

Objects Have Methods Associated with Them

`object.method()`

Use the period `.` to link the method to the object.

```
In [11]: string1 = 'Hello'

         string1 + '!'   # This is an operator. Operators combine objects in expressions.
         len(string1)   # This is a function. Functions take objects as arguments.
         string1.upper()   # This is a method. Methods are attached to objects.

Out[11]: 'HELLO'
```

String Methods: Formatting

- `S.upper()` – change to upper case
- `S.lower()` – change to lower case
- `S.capitalize()` – capitalize the first word
- `S.find(S1)` – return the index of the first instance of input

In [15]:

```
print('Make me scream!'.upper())
x = 'make this into a proper sentence'
print(x.capitalize()+'.')

print('Find the first "i" in this sentence.'.find('i'))
```

```
MAKE ME SCREAM!
Make this into a proper sentence.
1
```

String Methods: `strip` and `replace`

- `S.replace(S1, S2)` – find all instances of input1 and change to input2
- `S.strip(S1)` – remove whitespace characters from a string (useful when reading in from a file)

```
In [16]: x = ' This is a long sentence that we will use as an example.\n'
         print(x.replace('s', 'S'))
         print(x.strip())
         print(x.replace(' ', ''))
```

```
 ThiS iS a long Sentence that we will uSe aS an example.

This is a long sentence that we will use as an example.
Thisisalongsentencethatwewilluseasanexample.
```

String Methods: `split` and `join`

- `S.split(S1)` – split the string into a list
- `S.join(L)` – combine the input sequence into a single string

In [17]:

```
x = 'this is a collection of words i would like to break it into tokens'
y = x.split()    # default is to split on ' '
print(y)
print(x.split('o'))

x_new = '-'.join(y)
print(x_new)
```

```
['this', 'is', 'a', 'collection', 'of', 'words', 'i', 'would', 'like', 'to',
 'break', 'it', 'into', 'tokens']
['this is a c', 'llecti', 'n ', 'f w', 'rds i w', 'uld like t', ' break it in
t', ' t', 'kens']
this-is-a-collection-of-words-i-would-like-to-break-it-into-tokens
```

Unordered Types vs. Sequences

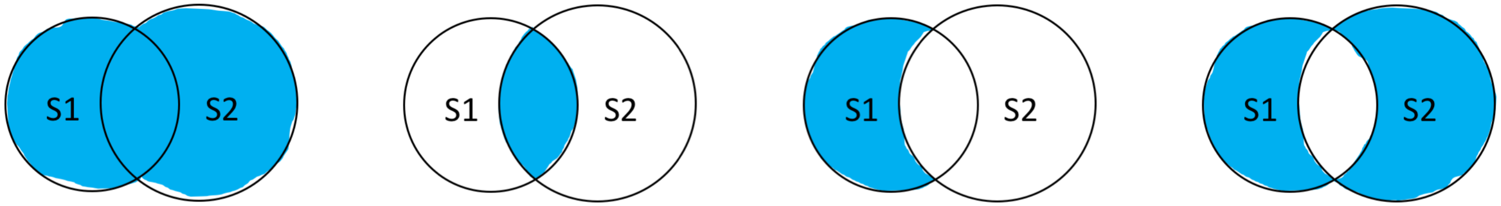
- Unordered types: `set`, `dict`
- Ordered (sequence) types: `str`, `list`, `tuple`

In [9]:

```
st = {1, 2, 2, 'a', 'b'} # sets are unordered
print(st)
```

```
{'b', 1, 2, 'a'}
```

Set Methods



- `S1.union(S2)`, `S1|S2` — elements in S1 or S2, or both
- `S1.intersection(S2)`, `S1&S2` — elements in both S1 and S2
- `S1.difference(S2)`, `S1-S2` — elements in S1 but not in S2
- `S1.symmetric_difference(S2)`, `S1^S2` — elements in S1 or S2 but not both

In [19]:

```
st1 = set('homophily')
st2 = set('heterophily')
print(st1 ^ st2)
```

```
{'m', 'e', 't', 'r'}
```

Dictionary Operations: Indexing

- Dictinaries are indexed by keys

```
In [20]: mydic = {'Howard': 'aerospace engineer', 'Leonard': 'physicist', 'Sheldon': 'physi
         cist',
                  'Penny': 'waitress', 'Raj': 'astrophysicist'}
         print(mydic['Raj'])
```

```
astrophysicist
```

Sequence Operations: Indexing

- Lists, tuples, and strings are indexed by numbers

```
In [21]: 'ABCDEFG'[2]

Out[21]: 'C'
```

Indexing in Python starts from 0!

Sequence Operations: Indexing

- Use `elem[index]` to extract individual sub-elements

```
In [22]: print( 'abc'[0] )
         print( (1,2,3)[-1]) # use negative numbers to index from the end
         print( [1,2,3][3])
```

```
a
3

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-22-50c4364c39d2> in <module>()
      1 print( 'abc'[0] )
      2 print( (1,2,3)[-1]) # use negative numbers to index from the end
----> 3 print( [1,2,3][3])

IndexError: list index out of range
```

Sequence Operations: Slicing

- Use `elem[start:end]` to get sub-sequence starting from index `start` and ending at index `end-1`

In [23]:

```
ls = [1,2,3,4,5]
print( ls[1:4] )
print( ls[:3] )
print( ls[1:] )

assert(ls[:]==ls[0:len(ls)])
```

```
[2, 3, 4]
[1, 2, 3]
[2, 3, 4, 5]
```

Sequence Operations: Extended Slices

- Use `elem[start:end:step]` to get sub-sequence starting from index `start`, in steps of `step`, ending at index `end-1`

In [35]:

```
ls = [1,2,3,4,5]
print( ls[::2] ) # get elements with even indeces
```

```
[1, 3, 5]
```

More Sequence Operations

In [24]:

```
tup1 = 3*(1,) # Notice that tuple of length 1 needs comma!
tup2 = tup1 + (2,2) # Concatenate the two elements
print(tup1, tup2)

print( max(tup2) ) # or min()
print( sum(tup2) )
print( tup2.count(1) )
print(tup2.index(2))
```

```
(1, 1, 1) (1, 1, 1, 2, 2)
2
7
3
3
```

- Why use tuples?
  - They use less memory than lists
  - They can be used as dictionary keys; lists can't

Mutability

- Immutable types: `str`, `tuple` (as well as all scalars)
- Mutable types: `list`, `set`, `dict`

Objects of mutable types can be modified once they are created.

In [25]:

```
dic = {1:'a', 2:'b'}
dic[3] = 'c'
print(dic)

ls = [5, 4, 1, 3, 2]
ls.sort()
print(ls)
```

```
{1: 'a', 2: 'b', 3: 'c'}
[1, 2, 3, 4, 5]
```

Mutability Can Be Quite Convenient

There are several useful list methods, see http://docs.python.org/3/library/stdtypes.html#mutable-sequence-types (http://docs.python.org/3/library/stdtypes.html#mutable-sequence-types):

- `L.append(e)`
- `L.insert(i, e)`
- `L.remove(e)`
- `L.extend(L1)`
- `L.pop(i)`
- `L.sort()`
- `L.reverse()`

In [26]:

```
L1 = [1, 2, 3]
L1.append(4)
print(L1)
L1.extend([5, 6, 7, 8, 9, 10])
print(L1)
```

```
[1, 2, 3, 4]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Mutability Can Also Be Dangerous

In [12]:

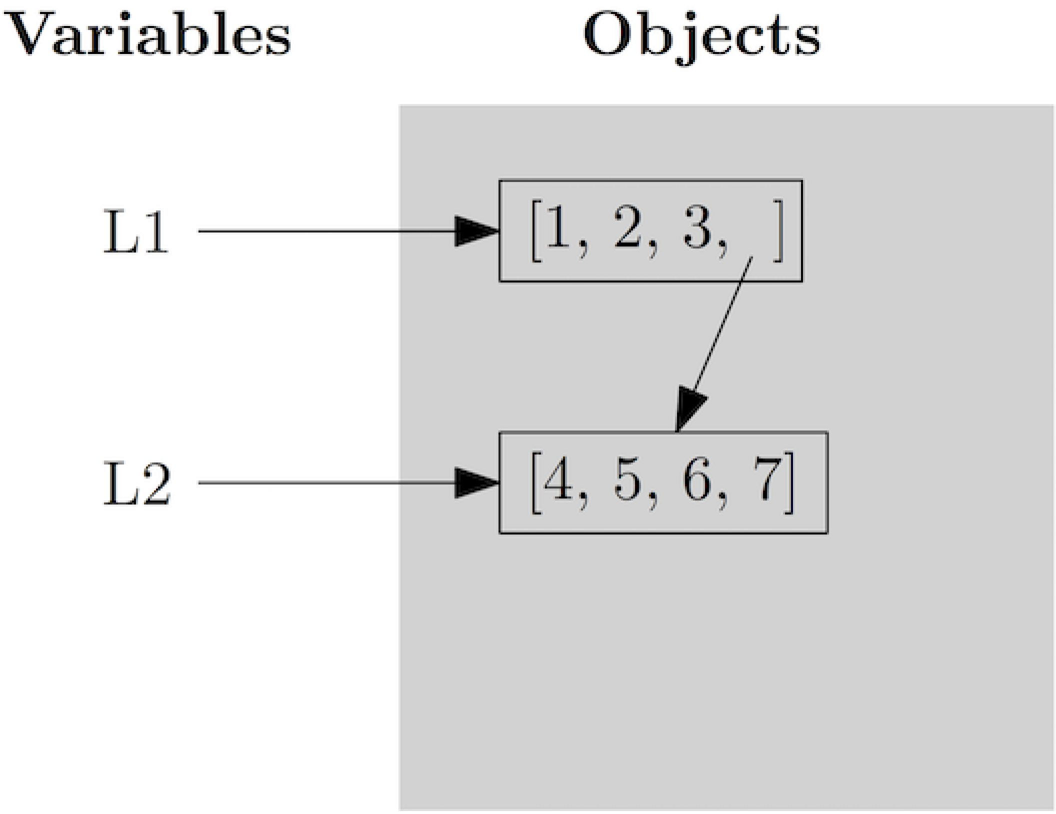
```
L1 = [1, 2, 3]
L2 = [4, 5, 6, 7]

L1.append(L2)
print(L1)

L2.extend([8, 9, 10])
print(L1)
```

```
[1, 2, 3, [4, 5, 6, 7]]
[1, 2, 3, [4, 5, 6, 7, 8, 9, 10]]
```

Aliasing vs. Cloning



In [28]:

```
L1 = [1, 2, 3]
L2 = L1[:]  # Using [:] is one way to clone

L1.reverse()
print(L2)
```

```
[1, 2, 3]
```

List Methods: `append` vs. `extend`

In [29]:

```
mylist = [1, 2, 3, 4]
mylist.append(5)
print(mylist)

mylist.extend([8, 7, 6])
print(mylist)
```

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

List Methods: `remove` vs. `pop`

In [13]:

```
mylist = [1, 2, 3, 4]

mylist.remove(1)
print(mylist)

popped = mylist.pop(1)
print(popped, mylist)
```

```
[2, 3, 4]
3 [2, 4]
```

List Methods: `L.sort()` vs. `sorted(L)`

In [31]:

```
mylist = [4, 5, 2, 1, 3]
mylist.sort()  # Sorts in-place. It is more efficient but overwrites the input.
print(mylist)

mylist = [10, 9, 6, 8, 7]
sorted(mylist)
print(mylist)

newlist = sorted(mylist)  # Creates a new list that is sorted, not changing the or
iginal.
print(mylist, newlist)
```

```
[1, 2, 3, 4, 5]
[10, 9, 6, 8, 7]
[10, 9, 6, 8, 7] [6, 7, 8, 9, 10]
```

Data Types in Python

Type	Scalar	Mutability	Order
`int`	scalar	immutable	
`float`	scalar	immutable	
`bool`	scalar	immutable	
`None`	scalar	immutable	
`str`	non-scalar	immutable	ordered
`tuple`	non-scalar	immutable	ordered
`list`	non-scalar	mutable	ordered
`set`	non-scalar	mutable	unordered
`dict`	non-scalar	mutable	unordered

- Objects have types
- Objects have methods

- **Lab**: Lists, lists, lists (and some strings)
- **Next week**: Control flow in Python