Intro to Python

Session A

“The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code -- not in reams of trivial code that bores the reader to death.”

–Guido van Rossum

Course Outline

- General overview of how Python works
- Interactive mode
- Data types and containers
- Anatomy of a script
- Parsing command line options
- Control flow and functions
- Importing modules
- Exception handling

Why Python

Where does it fit in?

Interpreted Languages

- Perl, PHP, Ruby, Shells (bash, csh, sh…), TCL
- Code is high level, easy to code
- Interpreted at runtime, translated to machine code
- Portable (ish)
- Powerful in their domains (perl->files, php->www), but generally slow and not as powerful as…

Compiled Languages

- C, C++, Swift, Fortran…
- Often difficult to program, 3-5 times as much code
- Memory management, header declarations
- Compiled for a specific isa/platform
- Not very portable
- Super fast, runs on metal
- Super mature, fully developed libraries allowing development of native apps

Python

- Faster than interpreted, slower (in some cases) than compiled
- No compile step, but compiled into byte code at runtime then cached
- Platform independent (when Python installed)
- Main advantage is integration of compiled, optimized libraries
- Native data structures all written in C
- Huge libraries of math, science & statistics functions precompiled in C and Fortran (numpy, scipy)
- Can write own precompiled modules

Python

- Fantastic ‘glue’ to integrate highly optimized code with an easy to program language
- Simple, elegant & easy to learn
- Designed for explicit rather than implicit, readability vs shorthand
- Faster than interpreted, slower than compiled although heavy work is generally run in precompiled modules (fast)
- Great blend of ease of use, portability, powerful libraries & speed
- Drawbacks? Modules must be installed to be used, dynamic typing, Python 2to3 backward compatibility saga

Interactive Mode

Exercise

- Create variables for each family member for their age (john=47)
- Create a tuple with the variables
- Print each persons age in one statement using placeholders
- Print the average age
- Print the average age of every 2nd person in family tuple

References

- sum([tuple]) len([tuple])
- >>>print(“%s spam %s” % [tuple])

```
>>> john=47
>>> alison=48
>>> zach=16
>>> reva=14
>>> family=(john,alison,zach,reva)
>>> family
(47, 48, 16, 14)
>>> print("John is %s, Alison is %s, Zach is %s, Reva is %s." % family)
John is 47, Alison is 48, Zach is 16, Reva is 14.
>>> print("Average age is %s" % (sum(family)/len(family)))
Average age is 31.25
```

```
>>> family[::2]
(47, 16)
>>> print("Average age of John and Zach is %s" % (sum(family[::2])/
len(family[::2])))
Average age of John and Zach is 31.5
>>>
```

Data Types

- Essentially all data structures in Python are objects. They have properties and actions.
- All ‘variables’ are references/pointers to objects (names for values)
    - ✳ When the reference is re-assigned, the type is reassigned
- An object (value) can have more than one name point to it
- Python uses dynamic strong typing
- The methods that can be run on a variable are determined at runtime.
- Native Python core data types are optimized c code, very efficient.
- Two basic types, mutable and immutable.

Data Types

- Immutable: numbers, strings, tuples, booleans, datetime

- Mutable: list, dict, set and variants.

Immutable

- Integer, float, fixed precision decimal, complex numbers, rationals… + 3rd party custom types
- Strings
- Tuples
- Booleans: False, None, zero of any numeric type, empty string, sequence or mapping(‘’,(),[],{}) are all False. Anything else is True
- datetime.datetime, datetime.date, datetime.time, datetime.timedelta

Mutable collections

- List
- Dictionary
- Set
- Collections (namedtuple, deque, ChainMap, Counter, OrderedDict, defaultdict…)

List

- Sequence similar to a tuple, but mutable.
- Accessed by index or slice: x[0] x[0:5]…
- Created with brackets (instead of parens)
- x=[1,2,3,4]
- x=[i for i in range(10)]

List Comprehension

- Filtered list
- New list =[expression for item in list if conditional]

```
>>> letters='AbCDefGH'
>>> lower=[char for char in letters if char.islower()]
>>> lower
['b', 'e', ‘f’]
```

- Access subsets through slices

```
>>>letters[:2]
Ab
```

Dictionaries

- Dictionaries can store arbitrarily large numbers of key->value pairs.
- Value can be anything (string, number, list, dictionary, custom objects)
- Very optimized efficient key->value lookup
- Stored as a hash table
- Not ordered, in fact order may change after adding element.
- Keys must be hashable (string, int, tuple)

Dictionaries

- d={}
- d={‘key’:value, ‘key2’:value2…}

```
>>> d={'site':'spo', 'lat':-89.908, 'lon':-24.8}
>>> d
{'site': 'spo', 'lat': -89.908, 'lon': -24.8}
>>>d[‘site’]
‘spo’
```

- d[key]=value

```
>>> d['num']=113
>>> d
{'site': 'spo', 'lat': -89.908, 'lon': -24.8, 'num': 113}
```

- d={key:value for key, value in items if conditional}

```
>>> lats={code:lat for code,lat,*t in sites}
>>> lats
{'asc': -7.9667, 'alt': 82.45, 'car': 40.63, 'spo': -89.98}
```

Dictionary Values

- print(lats[‘spo’])

- Loop through all

```
>>> for code,lat in lats.items(): print(code,lat)
...
asc -7.9667
alt 82.45
car 40.63
spo -89.98
```

- To get a value when you don’t know if key exists use get(key,default=None)

```
>>> lats.get("alt")
82.45
>>> lats.get("spam")
>>> lats.get("spam",0)
0
```

- To get a value when you don’t know if key exists and set a default value if not, use setdefault(key, setdefault=None) 82.45

```
>>> lats.setdefault("spam")
>>> lats.setdefault(“spam",0)
0
```

Sorting

- sorted([dict], reverse=False, key=None)

Dictionary Exercise

- Create a dictionary and populate with family members and their age
- Print out in order of age (google sort python dict by value)

```
>>> x={'j':47, 'a':48, 'z':16, 'r':14}
>>> sorted(x,key=x.get)
['r', 'z', 'j', 'a']
```

Sets

- Like a list, but unique values.
- Values must be hashable
- Very quick for determining membership
- Also good for uniqueifying a list
- s=set([list])

Collections

`collections` — Container datatypes

Source code: Lib/collections/__init__.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, `dict`, `list`, `set`, and `tuple`.

`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values
`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

Some Gotcha's

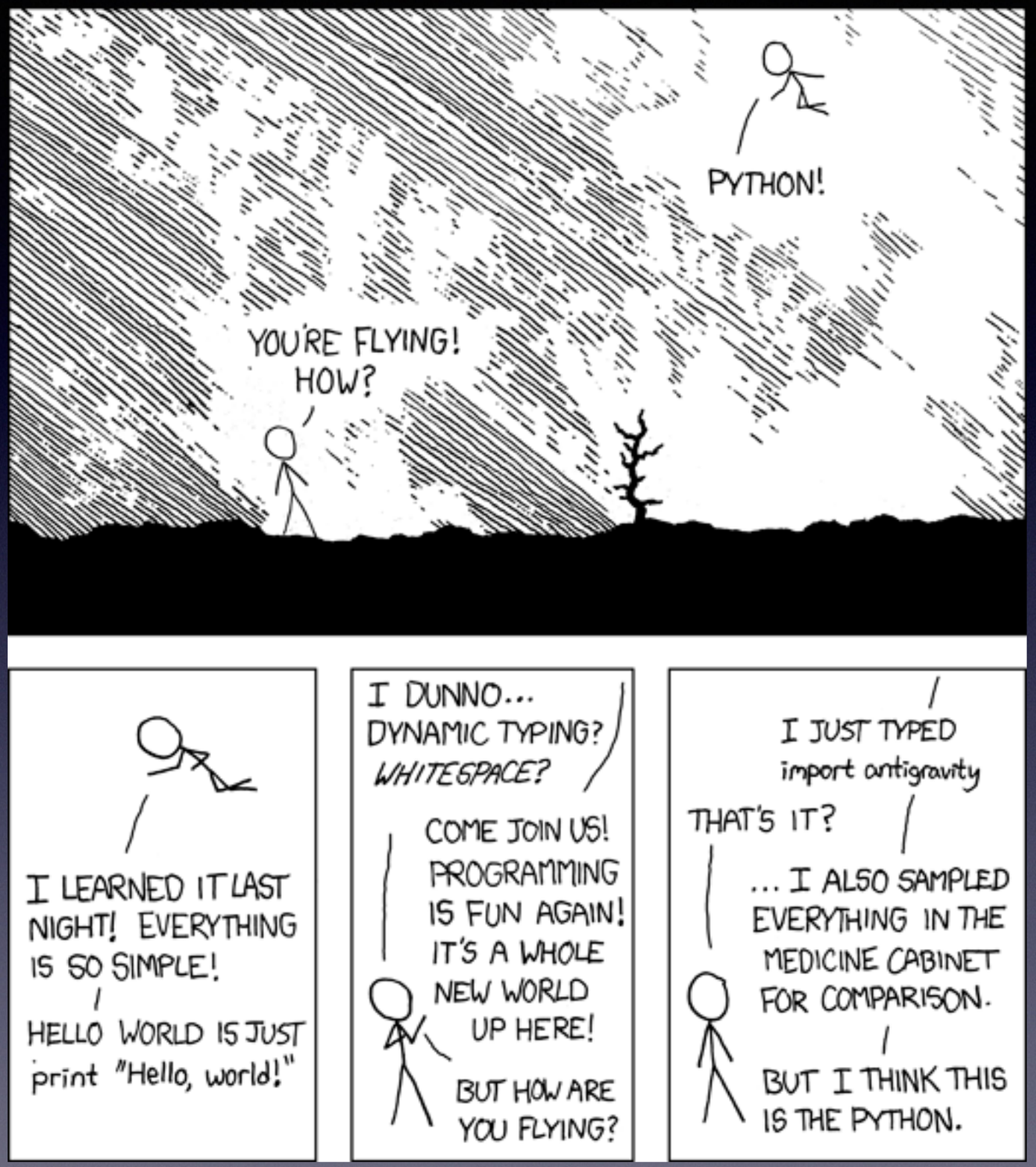
- Shallow copy [:] vs copy.deepcopy()
  ->everything is a pointer

- See https://nedbatchelder.com/text/names.html for a great explanation of python variables

- White space matters -> Tabs != Spaces

Scripts

XKCD


YOU'RE FLYING! HOW?
PYTHON!
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.

Exercise

- Make a copy of pythonTemplate.py and use to make your own siteData.py
- Add 2 additional arguments; start_date, end_date