Python Programming

Lecture 1

January 25th

Welcome

- Where are you?
  - Levine 100, Wu & Chen Auditorium
  - 2/8 we will be in Heilmeir Hall (Towne 100)
- Who am I?
  - Adam Aviv, 3rd year Ph. D. Student
    - ***aviv AT-SIGN cis.upenn.edu***
    - ***OH: Mon. 3-3:30 PM, Wed. 2-3 PM***
  - I study Security, Networking, and Distributed Systems
- Who are you?

Teaching Staff

- Kyle Super (TA)
  - Office: Moore 102
  - Office Hours: TBD
  - super *at-sign* seas.upenn.edu
- Karen Tao (Grader)
  - taot *at-sign* seas.upenn.edu
- Jaewoo Lee (Grader)
  - jaewoo *at-sign* seas.upenn.edu

Class Goals

- Become a capable Python programmer
  - types, objects, and the standard library
  - Networking, Regular Expression, 3rd party modules
- I do not expect you to be prolific
- I do not intend to teach Computer Science
  - but, various CS topics will be covered in examples and homeworks
- To never program in Perl ever again!
  - really!

My Prerequisite Expectations

- You know what a program is
- You have programmed before in some language
- You have some understanding of programing control flow, structure, variables and functions
  - Object oriented programming experience, is not required
- You have a general understanding of how a computer operates

Administrative

- Class Web Site
  - https://www.cis.upenn.edu/~cis192
- No Tests! No Curve!
  - 70% 5 HW's
    30% 1 Group Project
    10% Extra Credit
- Mark Lutz, *Learning Python,* O'REILLY
  - Free on Safari Online
- Guido van Rossum, *Python Library Reference*
  - Version 2.6 but not 3.0

Late Policy

- All HW due at **10pm** on the date specified
- 25% off after 1 day late
  - Submission will not be accepted more the 1 day late
- 50% off extra credit

Cheating Policy

- Don't do it!
- All programs should be your own work
  - I will use MOSS
- You may discuss programming paradigms, but the code should be your own.
- Ask, don't assume anything.
- Very low tolerance for cheating
  - **Zeros** on the assignment, report to **Office of Student Conduct**, **failing the class**, etc.

WARNING

This is a Programming Class

Bugs!

9/9

0800 Arctan started
1000 " stopped - arctan ✓
{ 1.2700 9.037 847 025
9.037 846 ?95 correct
13"u/c (032) MP - MC ~~2.130476415 (033)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 11.00 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test

1545 Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1630 Arctangent started.
1700 closed down.

Programming Insight

If debugging is the process of removing bugs, then programming must be the process of putting them in.

Edsger Dijkstra

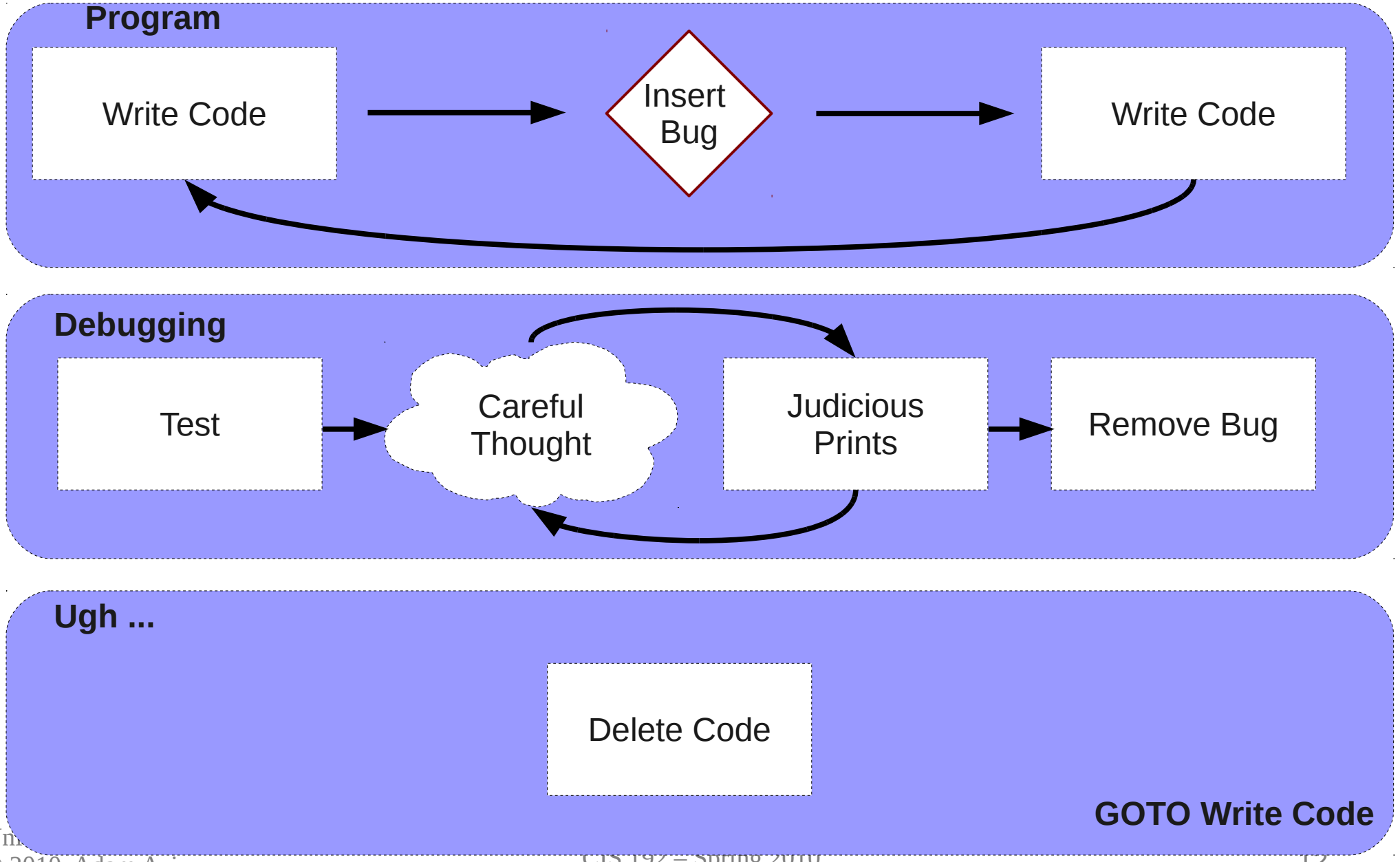
The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

Brian W. Kernighan

Deleted code is debugged code.

Jeff Sickel

Execute Until Converge



Moral ...

- Programming is HARD!
  - Provably hard.
- You are going to get frustrated, angry, resentful, and aggressive towards your computer.
- But! When you get it right ... it's the best.
- Ask for help, everyone makes mistakes.

Python

- Cover Python 2.* branch (www.python.org)
  - You should use 2.6*, but don't use 3.0
- Runs on Linux, Mac and Windows machines
  - You may use whatever OS you want
  - But, for best functionality and experience use Linux
- All code will be graded on the eniac cluster

Getting and Developing w/Python

- Linux (Fedora, Ubuntu, SUSE, etc.)
  - Already there, or install using package manager
  - Emacs, pico, IDLE, eclispe, etc.
- Windows
  - Download windows installer
  - Eclipse,IDLE
- Mac
  - Suggest Mac BSD ports, or standard installer
  - Emacs, default text editor, eclipse, etc.

When Submitting

ONLY TURN IN THE CODE

NO META-DATA

What is Python?

- Many faces
  - interpreted, scripting, object oriented
- History
- Future
- Why Python?
- Why not Python?
- Runs on all platforms (Windows, Linux, Mac)
- Demonstration Time!

Hello World

- The first program

```
print “Hello World”
```

- That's it

Running a Python Program

- The interpreter

```
>>>print "Hello World"
Hello World
```

- File

```
#!/usr/bin/python
# hello.py

print "Hello World"
```

```
$>python hello.py
Hello World
$>chmod +x hello.py
$>./hello.py
Hello World
```

The interpreter is your friend

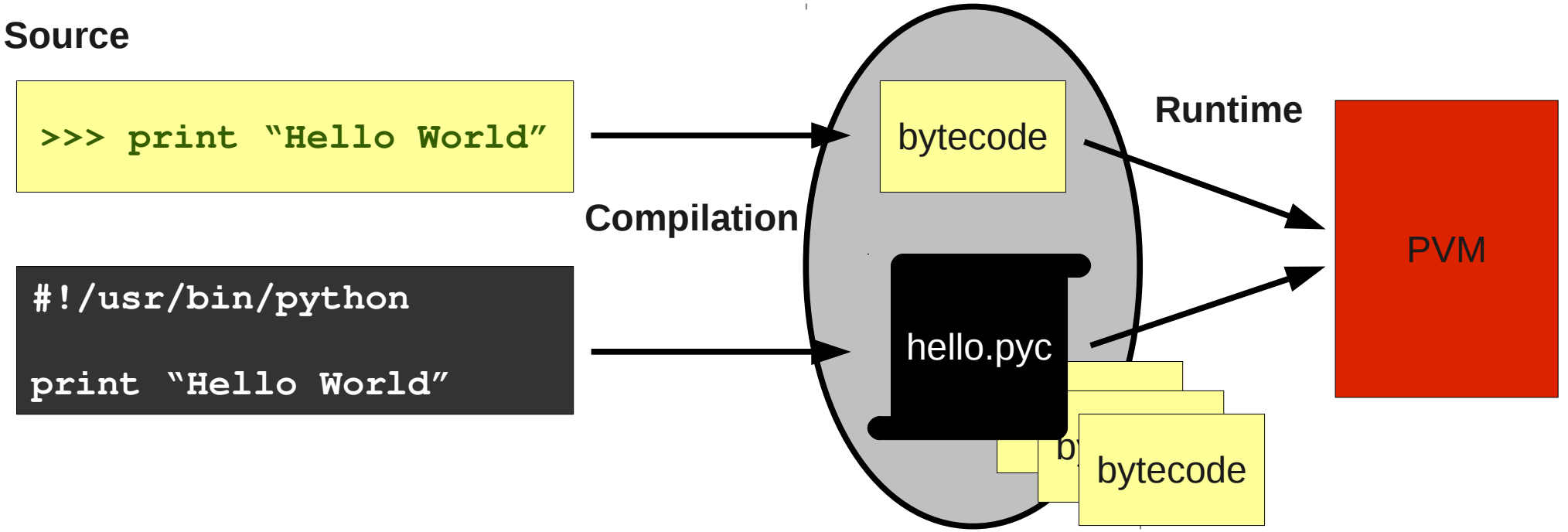
- Use the built in documentation
  - everything you need is in the terminal
- Don't know how a function works
  - do it in the terminal
- Want to try something
  - do it in the terminal
- Want to make Mac and Cheese
  - Use the stove-top or microwave, not the terminal
  - Everything else, use the terminal!

In Your Homework

- You use a file
  - ending in .py
- Should be runnable/executable
  - % ./myfile.py
  - % python myfile.py
- The terminal is an aid, use it as such

How Python Works

- Python Virtual Machine
  - Statements compiled into bytecode
  - bytecode executed by PVM



- What are the performance implications?

Code Organization
Modules

- Collections of code in one file
  - Or in a package (more in later lectures)
- Python Library comes with many, many modules
- Pull stuff in via `import` procedure
  - Access module via the . (dot) operator
- Allows for extendibility

import

- Code in file **`mymodule.py`**
- **`import mymodule`**
  - **`mymodule.myfunc(arg)`**
- **`from mymodule import myfunc`**
  - **`myfunc`** accessible directly
- **`from mymodule import *`**
  - All parts of module accessible directly

Garbage Collection

- Reference counter
  - Each variable that reference data-type increments a counter
  - When counter is zero, memory is remove
- Data can persist beyond intention
- Causes some 'gotchas'

Typing

- How do you type in Java and C?
  - You declare the type and set the value according
    - `int a,b; a = 1;`
- Python takes the inverse approach
  - You assign the variable a value, and Python gives the variable a type
    - `a = 1, b="10"`
- So, when does type checking occur?
- What if you just have to have a certain type?

Basic Types

- Numbers
  - integers, floats, complex
- Sequences
  - mutable vs immutable
  - lists, tuples, strings, sets ...
- Dictionaries
  - key value pairs
- None
  - the `None` type, like Null or NULL

Numbers

- Integers
  - what you think (`+ - / * ** %`)
- Floats, Longs
  - what you think, numbers with decimal calculations
- Operations
  - math module (**`sin cos`** constants)
    - **`from math import *`**
  - Bit-wise Operators (`| & ^ >> <<`)
- Do examples, in terminal

Augmented Operators

- `a += b -->  a = a + b`
- `a *= b -->  a = a * b`
- `a <<= b --> a = a << b`

- Works on things other then integers
- ```
  a = "Hello"
  a += " World"
  a == "Hello World"
  ```
  ```

Sequences

- The most important structure
  - Implicitly ordered
  - Iterable (used in for loops)
- Accessing vis [ ] operator
  - list[i], tuple[i], string[i]
- Immutable vs Mutable
  - `tuple[i] = 10` : Error
  - `string[i] = "a"` : Error
  - `list[i] = 10` : OK

Indexing in Sequences

- Has a length, `len()`
  - `IndexError: list index out of range`
- The colon operator for partial selection

```
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[0]
0
>>> a[0:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[1:]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[:6]
[0, 1, 2, 3, 4, 5]
>>> a[2:6]
[2, 3, 4, 5]
```

Lists

- Like arrays, but dynamic
  - Ordered/Indexed
  - Extendible/Reducible
    - `append()` `pop()` `remove()` `insert()` `+` `+=` `*`
- Elements can be anything, including lists
- Some operations happen in place
  - `sort()`,`reverse()`
- Read reference manual for complete list

Lists (cont)

```
>>> a = [1,2,3,4]
>>> type(a)
<type 'list'>
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
>>> a.pop()
5
>>> a
[1, 2, 3, 4]
>>> a[0] = 10
>>> a
[10, 2, 3, 4]
>>> a.reverse()
>>> a
[4, 3, 2, 10]
>>> a.sort()
>>> a
[2,3,4,10]
>>> a = list((1,2,3))
>>> a
[1,2,3]
```

```
>>> a = ["1",2,3,"hello"]
>>> a.index("hello")
3
>>> a.index(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
>>> a.remove("1")
>>> a
[2, 3, 'hello']
>>> a.count(2)
1
>>> a += [2]
>>> a.count(2)
2
>>> len(a)
4
>>> a
[2, 3, 'hello', 2]
>>> a == [2,3,2]
False
```

Tuples

- Set off by parenthesis or commas
- like lists, but immutable
- +, * operators still work
  - ~~`(1)+(2)==3`~~ `(1,)+(2,) == (1,2)`
  - ~~`(1)*(2)==2`~~ `(1,)*(2,) == (1,1)`
- Do assignment with tuples
  - `a,b = 1,"Hello"`
  - `a == 1  b == "Hello"`
  - `c = b,a` What is c?

Tuples (cont)

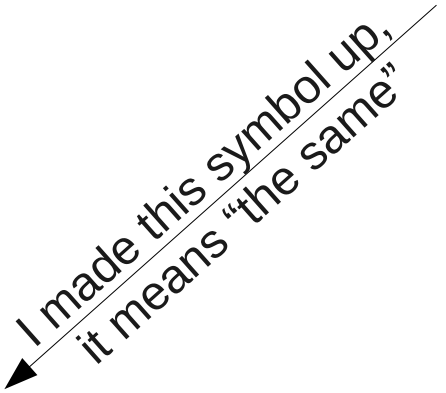
```
>>> a = (1,2,3)
>>> type(a)
<type 'tuple'>
>>> a[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> one,two,three = a
>>> one,two,three
(1, 2, 3)
>>> one
1
>>> two
2
>>> three
3
>>> a = 1,2,3
>>> a
(1,2,3)
>>> a = ()
>>> type(a)
<type 'tuple'>
```

Why use tuples?

- Efficient back end representation
- What does this do?

```
>>> x = 1
>>> y = 2
>>> x,y = y,x
```

- argument passing
  - `func(*(1,2,3)) ~=~ func(1,2,3)`
- quick bindings
  - `for k,v in dict.items()`



Strings

- set off by quotes
- lots of useful functions
  - split(), join(), +, +=
- Immutable

```
>>> a = "Hello World"
>>> b = 'Hello World'
>>> a == b
True
>>> a == """Hello World"""
True
```

```
>>> "spam, eggs hello world".split()
['spam,', 'eggs', 'hello', 'world']
>>> "spam, eggs hello world".split(",")
['spam', ' eggs hello world']
```

```
>>> "A".lower()
'a'
>>> "a".upper()
'A'
```

```
>>>",".join(("spam","eggs"))
'spam,eggs'
>>>"!".join(("spam","eggs")))
'spam!eggs'
```

String Formating

- Like in other languages
  - Set off by format marker % within string
  - Arguments come following string and % marker

```
>>> exclamation = "Ni"
>>> "The knights who say %s!" % (exclamation)
'The knights who say Ni!'
```

- %d, %f, %s, %x, %%

```
>>> "%f %s %d you" % (0.5,"spam",4)
'0.500000 spam 4 you'
>>> "%0.2f %s %d you" % (0.5,"spam",4)
'0.50 spam 4 you'
```

sets

- ....
- Actually you'll learn this for yourself in the HW
  - Read the Library Reference

Dictionaries

- Key to value binding
  - Give it the key, it gives you the value
  - Give it a value and a key, it stores it for you
- Keys must be immutable (why?)
  - strings, ints, numbers, tuples, objects, but not lists!
  - **`keys()`,`values()`,`items()`,`has_key()`**
- Create a new key by assignment or **`set()`**
- Remove a key using **`del()`** function or **`pop()`**

Dictionaries (continued)

```
>>> a = {"1":1, 2:"2"}
>>> a[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> a["1"]
1
>>> a[3] = "Hello"
>>> a
{"1":1, 2:"2", 3:"Hello"}
>>> del(a['1'])
>>> a
{2:"2", 3:"Hello"}
>>> a.keys()
["1",2,3]
>>> a.values()
[1.5,"2","Hello"]
```

Casting

- Move from one type to another
- Can have errors
  - some casting require specific structures
- resulting form, not always what you expect

```
>>> int("1")
1
>>> float(int("1"))
1.0
```

```
>>> tuple([1,2])
(1, 2)
>>> list(tuple([1,2]))
[1, 2]
```

```
>>> str([1,2,3])
'[1, 2, 3]'
>>> str((1,2,3))
'(1, 2, 3)'
>>> str({"1":1, "2":2})
"{'1': 1, '2': 2}"
```

```
>>> dict([("1",1),("2",2)])
{'1': 1, '2': 2}
>>> dict([["1",1],["2",2]])
{'1': 1, '2': 2}
>>> dict(a=1, b=2)
{'a': 1, 'b': 2}
```

Boolean

- Can't use: `||` `&&` `~`
- Use: **`and or not`**
  - Why this choice? Don't know.
- **`True`** and **`False`** basic Boolean types
- 0 and 0.0 is **`False`**
- empty sequence is **`False`**
- empty dictionary is **`False`**
- Everything else is **`True`**

Comparisons

- Equivalent
  - `==`
  - `=` is assignment not “is equal?”
- Non-equivalent
  - ~~`!=`~~ `not a == b`
- Less/Greater Then
  - `<` `>`
- Less/Greater Then or Equal
  - `<=` `>=`

type checking

- Most time, you don't care, but sometimes you do
- **`type()`** - returns the *type object*
- Use the **`is`** operator over ==

```
>>> type([]) is list
True
>>> type([]) is dict
False
>>> a = [1,2,3]
>>> type(a) == type([])
True
```

- Both work, then why is one better?
- Different paradigm for objects and sub-types

Control Flow

- Just like other languages
  - `if proposition:` ◄ *Colon!*

Indention►
```
    <code block>
elif proposition:
    <code block>
else:
    <code block>
```
  - `while proposition:`
```
    <code block>
```
  - `for i in sequence:`
```
    <code block>
```
- Blocking done using White Space
  - **Indention is very important**

Control Flow (cont)

- For loops work over sequences / iterables

```
for i in range(10):
   print i,
```

- While loops

```
i = 0
while not i == 10:
   print i,
   i+=1
```

- **`if`**, **`elif`**, **`else`** blocking

```
if a == b:
   print "b"
elif a == c:
   print "c"
else:
   print "wtf?"
```

For Loops Example

```
>>> for a in "hello world":
...     print a
...
h
e
l
l
o

w
o
r
l
d
```

```
>>> for a in xrange(10):
...     print a,
...
0 1 2 3 4 5 6 7 8 9
>>> for a in range(10):
...     print a,
...
0 1 2 3 4 5 6 7 8 9
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> xrange(10)
xrange(10)
```

What is the difference between range() and xrange()?

(for home work)

`break` **and** `continue`

- Loop Control
- Break out of the loop
- Continue the loop
- Scoped, only works on most inner loop

```
for i in range(10):
    for j in range(i):
        if i > 5:
            break
        print i,j
```

```
i = 0
while True:
    if i > 10:
        break
    elif i%2 == 0:
        continue
    print i
    i += 1
```

More On Control Flow

- `pass`
  - Don't do anything, makes white-spacing happy
- No case/switch, use dictionaries (give ex.)
- For loops work over sequences
  - read: for each item in the sequence do this
    - Not really, over *iterables*
  - `for a in range(10)`
    - `a` is introduced/set in the local scope
    - `range(10)` converted to iterable
- While loops as you expect them

White Space Example

```
>>> a = 10
>>> while a >=0:
...     print a
...     a -= 1
...
10
9
etc.
```

White Space in Interpreter
STILL NEED TO INDENT

```
#!/usr/bin/python
# hello.py

#unested
a = 10
while a>=0:
print a
a -= 1

#nested
a = 10
while a>=0:
    print a
    for i in range(a):
        print i, "Hey"
    a -= 1
```

Functions

- **`def`**
- **`return`**

```
>>> def add2(x):
...     return x+2
...
>>> add2(10)
12
>>> add2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add2() takes exactly 1 argument (0 given)
```

- what if no return, return's **`None`**
- Also uses White-Space blocking

Documentation

- Python has built in documentation
- The **`__doc__`** string comes after a **`def`**, **`class`**, or beginning of a module

```
>>> def add2(x):
...     """add2 takes in a number and adds 2"""
...     return x+2
...
>>> print add2.__doc__
add2 takes in a number and adds 2
```

- Get to it through the "." operator
  - **`[].__doc__`**
- You should document in your code

dir()

- directory listing
- what are the members of this thing
  - returned as list, why might that be useful?

```
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'add2', 'b', 'p', 'p1',
'p2']
>>> dir("")
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__str__', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

help()

- The help function displays ... help information
- Basically the library reference in the interpreter

```
>>> help(str)
Help on class str in module __builtin__:

class str(basestring)
 |  str(object) -> string
 |
 |  Return a nice string representation of the object.
 |  If the argument is a string, the return value is the same object.
 |
 |  Method resolution order:
 |      str
 |      basestring
 |      object
 |
 |  ...
```

`__builtins__`

- There are a number of built in functions and variables, you can access them any time
- Stored in the module **`__builtins__`**
- To see what they are you can
  - **`dir(__builtins__)`**
    - This is a bit terse
  - **`help(__builtins__)`**
    - This is a bit verbose
  - Use the Library Reference
    - This is just right

Some very, very useful `__builtins__`

- **`zip()`** : join two sequences
  - **`zip(j,k) -> [(j[0],k[0]), (j[1],k[1]),...,(j[n],k[n])]`**
- **`map()`** : apply func. to a sequence in place
- **`reduce()`** : apply func. cumulatively to a seq.
  - **`reduce(multiply,range(10)) -> 3628800`**
- **`iter(), cmp(),repr() and str()`**
- **`bin(), chr(), ord(), hex(), abs()`**
- **`len(), max(), min()`**

Homework

- First HW out today!
  - Due next Wed. at 10pm
- Get started early
- Ask for help!

- ***See you next week***