

# Python

# What's This Course About? - Python 2.7

- Language Introduction
  - Installation, starting up, executing code, variables
  - Functions, modules, classes
  - Screen I/O, data types, control structures, file/socket I/O, exception handling
- Pythonic Stuff
  - Comprehensions, generators, iterators, context managers, decorators
  - Command line arguments, iterator tools, lambdas, functions-in-functions
  - Multiprocessing, utilities
- Useful Stuff
  - Unittest, cool Python API parts, code reviews

# Why?

- Python 2.7 is popular
- Python is expressive!
  - Its API is massive
  - Many other projects already do what you need
  - It's great for prototyping
  - It's popular for automating things like test
- Programmers are team members
- Programmers should write code that's easy to read and maintain

# Daily Structure

- 0800 - Lecture & Exercises
- 1100 - Labs & Lunch
- 1400 - Code Review
- 1500 - Code Fight Tournament

# Quick Poll

- Who here is good at Python?
- Who here has used Python some, but not much?
- Who here has no Python experience?

# Python Documentation

# Installation and Startup

- Download and install...
  - apt-get install python
  - OS X has it already, but homebrew will get you an updated version
  - <https://www.python.org/downloads/>
- Python package managers
  - easy\_install and pip
  - /usr/local/lib/python2.7/site-packages
  - pip install [whatever](#)
- Startup
  - python, python2, python2.7
  - idle, idle2, idle2.7

# Writing a Script

```
#!/usr/bin/env python
```

```
from __future__ import print_function
```

```
# Print out a sober hello world
```

```
print("Hello, world.")
```

```
def diag_print(msg):
```

```
    """Print a message on a diagonal
```

```
    :param str msg: The message to print
    """
```

```
    for ind, char in enumerate(msg):
        print(" " * ind + char)
```

```
if __name__ == '__main__':
```

```
    diag_print("SUP WORLD?!?!")
```

```
f@f ~/pcp17 % vi helloworld.py
```

```
f@f ~/pcp17 % chmod +x helloworld.py
```

```
f@f ~/pcp17 % ./helloworld.py
```

```
Hello, world.
```

```
S
```

```
U
```

```
P
```

```
W
```

```
O
```

```
R
```

```
L
```

```
D
```

```
?
```

```
!
```

```
?
```

```
!
```

```
f@f ~/pcp17 %
```





# Coding Style

- Python Enhancement Proposal (PEP) 8

- "code is read much more often than it is written" - "readability counts"
- "know when to be inconsistent" - "use your best judgment"
- "use 4 spaces per indentation level"
- "limit all lines to a maximum of 79 characters"

- Using Vim - ~/.vimrc

- :set shiftwidth=4 "vim's shifts using >> represent 4 characters
- :set tabstop=4 "tabs you type represent four character positions
- :set expandtab "turn any tab you type into spaces
- :set ruler "an awesome option that
- :set columns=79 "only display 79 characters per line
- :syntax on "syntax highlighting
- :filetype indent plugin on "turn on a plugin to do most of indenting based on filename

# Variables

```
>>> from __future__ import print_function
>>> a = 15
>>> b = 15.4
>>> c = [a,b]
>>> print(c)
[15, 15.4]
```

```
>>> a = 24
>>> print(c)
[15, 15.4]
```

```
>>> print( (a,b) )
(24, 15.4)
```

```
>>> print = "asdf"
```

```
>>> print( (a,b) )
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

- Variables are names with bindings
- Almost everything is an instance of a class
- Functions can be rebound...
- What types are visible here?

# Types

```
>>> a=[1, 2, 3.4, 4/7, 10/7, 5/7.0, "asdf"]
>>> print(a)
[1, 2, 3.4, 0, 1, 0.7142857142857143, 'asdf']
>>> for i in a:
...     type(i)
...
<type 'int'>
<type 'int'>
<type 'float'>
<type 'int'>
<type 'int'>
<type 'float'>
<type 'str'>
>>> type(True)
<type 'bool'>
```

"The principal built-in types are numerics, sequences, mappings, files, classes, instances and exceptions."

"Some operations are supported by several object types; in particular, practically all objects can be compared, tested for truth value, and converted to a string (with the repr() function or the slightly different str() function). The latter function is implicitly used when an object is written by the print() function."

Let's just talk through what we're seeing on the left...

# Integers...

```
>>> int_version = 9223372036854775807
>>> long_version = (2**63 - 1)
>>> type(int_version)
<type 'int'>
>>> type(long_version)
<type 'long'>
>>> int_version == long_version
True
>>> print(int_version, long_version)
9223372036854775807 9223372036854775807
>>> int_version, long_version
(9223372036854775807,
9223372036854775807L)
>>> int_version += 1
>>> type(int_version)
<type 'long'>
```

- Integers come in two forms - int and long
- The code on the left is from a 64 bit machine
- I have never run into an instance where this caused me a problem in practice...
- Python 3 has only one integer type and it always works

# Type Conversion

```
>>> int_v = 15
>>> float(int_v)
15.0
>>> int(float(int_v))
15
>>> int_v + .15
15.15
>>> int(int_v + .15)
15
>>> int(int_v + .6)
15
>>> str(int_v)
'15'
>>> str(float(int_v))
'15.0'
>>> int("15.5")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10:
'15.5'
>>> int("15")
15
```

- Everything is an instance of a class...
- Even numbers...
- What do you expect to happen when you provide a class constructor with a value?
- This kind of type conversion works in most places

# Sequence Types

```
>>> str = "this is a string"
>>> int_list = [1,2,3,1,2,3]
>>> int_tuple = (1,2,3,4)
>>> int_set = {1,2,3,1,2,3}
>>> str[0:4]
'this'
>>> str[:4]
'this'
>>> int_list[::2]
[1, 3, 2]
>>> int_tuple[2:]
(3, 4)
>>> int_set
set([1, 2, 3])
>>> list(int_tuple)
[1, 2, 3, 4]
>>> set(int_list)
set([1, 2, 3])
```

- We've got declarations!
- We've got slicing!
  - Implicit slicing indices...
  - Slice stepping!
- We've got type conversions!
- What's this set thing doing?

<https://docs.python.org/2/library/stdtypes.html#set-types-set-frozenset>

# Mutability

```
>>> int_list=[1,2,3,4]
>>> initial_id = id(int_list)
>>> initial_id
4491068768
>>> int_list.append(15)
>>> int_list
[1, 2, 3, 4, 15]
>>> final_id = id(int_list)
>>> initial_id == final_id
True
```

```
>>> foobar = 4
>>> initial_id = id(foobar)
>>> initial_id
140204414349520
>>> foobar = 5
>>> final_id = id(foobar)
>>> initial_id == final_id
False
```

- Previously modifying a variable cause the variable name to be reassigned
- Will that work well with lists?
- Consider changing only one element of the list...
- So - lists are mutable! Sets, dictionaries too, but not integers, strings, tuples...
- You cannot change just one letter in a string



# What about binary and unicode?

```
>>> a=list()
>>> for i in range(256):
...     a.append(chr(i))
...
>>> a
['\x00', '\x01', '\x02', '\x03', '\x04', '\x05', '\x06', '\x07', '\x08', '\t', '\n', '\x0b', '\x0c', '\r', '\x0e', '\x0f', '\x10', '\x11',
'\x12', '\x13', '\x14', '\x15', '\x16', '\x17', '\x18', '\x19', '\x1a', '\x1b', '\x1c', '\x1d', '\x1e', '\x1f', ' ', '!', '"', '#', '$',
'%', '&', "'", '(', ')', '*', '+', ',', '-', '.', '/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>', '?', '@', 'A', 'B', 'C', 'D',
'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '[', '\', ']', '^', '_', '`', 'a', 'b',
'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '|', '}', '~', '\x7f', '\x80',
'\x81', '\x82', '\x83', '\x84', '\x85', '\x86', '\x87', '\x88', '\x89', '\x8a', '\x8b', '\x8c', '\x8d', '\x8e', '\x8f', '\x90',
'\x91', '\x92', '\x93', '\x94', '\x95', '\x96', '\x97', '\x98', '\x99', '\x9a', '\x9b', '\x9c', '\x9d', '\x9e', '\x9f', '\xa0',
'\xa1', '\xa2', '\xa3', '\xa4', '\xa5', '\xa6', '\xa7', '\xa8', '\xa9', '\xaa', '\xab', '\xac', '\xad', '\xae', '\xaf', '\xb0',
'\xb1', '\xb2', '\xb3', '\xb4', '\xb5', '\xb6', '\xb7', '\xb8', '\xb9', '\xba', '\xbb', '\xbc', '\xbd', '\xbe', '\xbf', '\xc0',
'\xc1', '\xc2', '\xc3', '\xc4', '\xc5', '\xc6', '\xc7', '\xc8', '\xc9', '\xca', '\xcb', '\xcc', '\xcd', '\xce', '\xcf', '\xd0', '\xd1',
'\xd2', '\xd3', '\xd4', '\xd5', '\xd6', '\xd7', '\xd8', '\xd9', '\xda', '\xdb', '\xdc', '\xdd', '\xde', '\xdf', '\xe0', '\xe1',
'\xe2', '\xe3', '\xe4', '\xe5', '\xe6', '\xe7', '\xe8', '\xe9', '\xea', '\xeb', '\xec', '\xed', '\xee', '\xef', '\xf0', '\xf1', '\xf2',
'\xf3', '\xf4', '\xf5', '\xf6', '\xf7', '\xf8', '\xf9', '\xfa', '\xfb', '\xfc', '\xfd', '\xfe', '\xff']
>>> type(u"asdfasdf")
<type 'unicode'>
```

# Mappings

```
>>> foo={  
...     "a key": "a value",  
...     "key 2": 15.3,  
...     (2,1) : [1,2,3,4,5]  
... }
```

```
>>> type(foo)  
<type 'dict'>
```

```
>>> foo["a key"]  
'a value'
```

```
>>> foo[(2,1)]  
[1, 2, 3, 4, 5]
```

```
>>> foo["lucky number"] = 13  
>>> foo[0] = "zombo.com"
```

```
>>> foo  
{0: 'zombo.com', 'key 2': 15.3, 'a key': 'a value',  
(2, 1): [1, 2, 3, 4, 5], 'lucky number': 13}
```

- Dictionaries are `_the_` mapping type
- They are a hash map
- Every key must be unique, values don't have to be

# Mappings Continued

```
>>> foo[ ["a","b"] ] = 27
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: unhashable type: 'list'
```

```
>>> foo[0] = ["a","b"]
```

```
>>> foo
```

```
{0: ['a', 'b'], 'key 2': 15.3, 'a key': 'a value', (2, 1):
```

```
[1, 2, 3, 4, 5], 'lucky number': 13}
```

- "Unhashable" types cannot be dictionary keys
- A good rule of thumb - if a type is mutable then it is not hashable

# How does hashing work?

```
>>> foobar = "asdf"; hash(foobar)
-2835731962866600674
```

```
>>> foobar = 49; hash(foobar)
49
```

```
>>> foobar = 37.2; hash(foobar)
2107415142
```

```
>>> foobar = (15,); hash(foobar)
3430017387552
```

```
>>> foobar = [15,16,17]; hash(foobar)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

- What type is the builtin hash function returning?
- What happens when you hash something unhashable?
- How does the hash function know when something is hashable?
  
- You can often use a semicolon to put multiple things on one line

# HOW DOES IT KNOW!?!?!?

```
>>> dir((1,2))
['__add__', '__class__', '__contains__',
 '__delattr__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__',
 '__getnewargs__', '__getslice__', '__gt__',
 '__hash__', '__init__', '__iter__', '__le__',
 '__len__', '__lt__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rmul__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', 'count', 'index']
```

- The builtin dir function displays all attributes and functions of an instance
- Hashable instances have a `__hash__` function
- The hash builtin calls this function and returns the value
- Can you define `__hash__` when you create your own classes and instances?
- So - do list objects have a `__hash__`?

# HOW DOES IT KNOW!?!?!?!?

```
>>> dir([1,2])
['__add__', '__class__', '__contains__',
 '__delattr__', '__delitem__', '__delslice__',
 '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__getslice__',
 '__gt__', 'hash', '__iadd__', '__imul__',
 '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__reversed__',
 '__rmul__', '__setattr__', '__setitem__',
 '__setslice__', '__sizeof__', '__str__',
 '__subclasshook__', 'append', 'count', 'extend',
 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

- DANG IT! This list has a `__hash__`!
- HOW DOES THE HASH BUILTIN KNOW?!?!?!?!?

# Let's investigate...

```
>>> (1,2,3).__hash__  
<method-wrapper '__hash__' of tuple object at  
0x10baed910>
```

```
>>> (1,2,3).__hash__()  
2528502973977326415
```

```
>>> hash((1,2,3))  
2528502973977326415
```

```
>>> [1,2,3].__hash__
```

```
>>> [1,2,3].__hash__ is None  
True
```

```
>>> [1,2,3] is [1,2,3]  
False
```

```
>>> [1,2,3] == [1,2,3]  
True
```

- So - on the tuple, `__hash__` is a method-wrapper
  - This is some kind of fancy object-oriented hell they've done - it's a function
- When we call it, we get an int
- On the list, `__hash__` is "None"
- Wait - KARL - WHAT THE HECK IS "is" and "None" !?!?!?

# Time for some exercises...

For 15 minutes, on the Python console write...

- A for loop that calculates the first 20 numbers of the Fibonacci sequence
- A list of the primary colors red, blue, and yellow
  - Then, use functions on the list instance to add "purple", "green", and "orange" to their proper position between the primary colors
  - We did not discuss the function you need - look it up in the Python documentation

Also - take a break



# Solutions

```
>>> i, j = 0, 1
```

```
>>> for cnt in range(20):
```

```
...     i, j = j, i+j
```

```
...     print i
```

```
>>> colors=["red","blue","yellow"]
```

```
>>> colors.insert(1,"purple")
```

```
>>> colors.insert(3,"green")
```

```
>>> colors.insert(0,"orange")
```

```
>>> colors.append("orange")
```

```
>>> colors
```

```
['orange', 'red', 'purple', 'blue', 'green', 'yellow', 'orange']
```

If and while - control flow

# Control Flow: if/elif/else

```
bar = 10
```

```
foo = 5 if bar >= 10 else 10
```

```
if foo == 10:
```

```
    print('foo = 10')
```

```
elif foo == 5:
```

```
    print('foo = 5')
```

```
else:
```

```
    print('foo != 10 or 5')
```

- If, elif and else work how you'd expect...
- Watch indentation!
- Check out that awesome ternary operator!

# Control Flow - While

```
>>> foo_list = ["up", "you", "give",  
               "gonna", "never", "foo", "bar",  
               "bar", "foo", "foo", "bar"  
               ]
```

```
>>> while "foo" in foo_list:  
...     print foo_list.pop()  
...  
bar  
foo  
foo  
bar  
bar  
foo
```

- Does while work exactly how you hoped?
- Yes it does. Don't lie to me.
- Ok - so how about "in"? Why didn't we get Rick Rolled?
- How about "pop"? Pretty awesome, huh?

# Control Flow - Break, Continue, Pass

```
>>> known_primes = []
>>> for i in range(100):
...     if i == 0 or i == 1:
...         continue
...     is_prime = True
...     for prime in known_primes:
...         if i % prime == 0:
...             is_prime = False
...             break
...     if is_prime:
...         known_primes.append(i)
...     else:
...         pass
...
>>> print known_primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

- Continue skips execution of the rest of the iteration of the loop, moving to the next
  - Break ends all execution of the loop...
  - Pass is useful when Python requires an indented block, but you don't have one
    - For example, for stub code
    - Was there any reason for me to put in the else and pass here?
  - Why did we skip 0 and 1 at the beginning of the loop? Was there a better way to do that?
  - Are continue and break different than "goto"? Why is that good or bad?
- DISCUSS

# Functions, Modules, Classes

In here, we'll write them and talk about the basics of how Python programs are organized

# Functions

```
>>> def this_is_a_func(arg1, arg2, *args,
**kwargs):
...     print "arg1 was {}".format(arg1)
...     print "arg2 was {}".format(arg2)
...     print "args was {}".format(args)
...     print "kwargs was {}".format(kwargs)
...     return arg1 + arg2
...
>>> retval =
this_is_a_func(15,22,"first","second",key1="foo",
key2="bar")
arg1 was 15
arg2 was 22
args was ('first', 'second')
kwargs was {'key2': 'bar', 'key1': 'foo'}
>>> print retval
37
```

- Line indentation after the "def" must be consistent
- \*args becomes a list of positional arguments that were left over
- \*\*kwargs becomes a dictionary of keyword arguments that were left over
- But, it doesn't have to be this way...

# Functions

```
>>> def this_is_a_func(arg1, arg2, *args,
... **kwargs):
...     print "arg1 was {}".format(arg1)
...     print "arg2 was {}".format(arg2)
...     print "args was {}".format(args)
...     print "kwargs was {}".format(kwargs)
...     return arg1 + arg2
...
>>> retval = this_is_a_func(arg2=10, arg1=15,
foo="bar")
arg1 was 15
arg2 was 10
args was ()
kwargs was {'foo': 'bar'}
```

- Here - we've used keyword arguments out of order, and left args empty

NO PROBLEM MON!



# Classes

```
#!/usr/bin/env python
```

```
"""test.py"""
```

```
class my_base_class(object):  
    class_var = 20  
  
    def __init__(self, special_num):  
        self.special_num = special_num  
  
    def print_special(self):  
        print "My special num is {}".format(self.special_num)  
  
class child_class(my_base_class):  
    def __init__(self, special_num):  
        super(child_class, self).__init__(special_num * 2)  
  
if __name__ == "__main__":  
    ch = child_class(11)  
    ch.print_special()
```

- Base classes in Python 2 should inherit from "object"
- The `__init__` function is called like a constructor
- Self is the first argument in almost every case - this is a convention...
- How about multiple inheritance?
- "Super" is super confusing, but less-so in python 3...
- What will executing `./test.py` produce?

# Modules and Packages

- `mkdir my_module/`
- `mv test.py my_module/`
- `vi my_module/__init__.py`

```
from my_module.test import child_class
```

- `python2`

```
>>> import my_module
>>> my_module.child_class(22).print_special()
My special num is 44
>>> dir(my_module.child_class(22))
['__class__', '__delattr__', '__dict__', '__doc__',
 '__format__', '__getattribute__', '__hash__',
 '__init__', '__module__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', 'class_var',
 'print_special', 'special_num']
```

- Modules are files containing python code
- Every time we've written code in a file, that has created a module
- Packages are directories containing modules, including one named `__init__.py`
- `__init__.py` is usually empty, but if it's not empty, things imported into it are imported whenever the package is imported
- What are the two different ways to use import?

# EXERCISE!!!

Make a package called "school", with modules "high", "middle" and "elementary". Each module should have functions:

- Function "add\_class" that takes one argument, a string that names the class

- Function "list\_classes" that returns a list of all classes defined so far

- Property "name" that gives a school name that may be set

Create a script that demonstrates this functionality.

BONUS - also do this where high, middle and elementary are classes... Maybe "school" becomes a module instead of a package?

# EXERCISE!!!

```
% ls school
```

```
__init__.py  elementary.py  high.py  middle.py  
__init__.pyc elementary.pyc high.pyc middle.pyc
```

```
----- high.py, middle.py, elementary.py -----
```

```
class_list = list()  
name = ""
```

```
def add_class(class_name):  
    class_list.append(class_name)
```

```
def list_classes():  
    return class_list
```

```
----- init.py -----
```

```
import school.high as high  
import school.middle as middle  
import school.elementary as elementary
```

```
----- run_schools.py -----  
#!/usr/bin/env python
```

```
import school
```

```
if __name__ == "__main__":  
    class_names = ["english", "spanish", "opera",  
                  "math", "science"]  
    schools = {  
        "John Booker High": school.high,  
        "James Knight Middle": school.middle,  
        "Little Tykes Elementary": school.elementary,  
    }
```

```
for school_name, school_type in schools.items():  
    school_type.name = school_name  
    for name in class_names:  
        school_type.add_class(name)
```

```
for school_type in schools.values():  
    print "Classes at {}".format(school_type.name)  
    for course in school_type.list_classes():  
        print "\t{}".format(course)
```

# Comprehensions

List, dictionary, set... "join" on strings...

# Comprehensions

```
>>> [i*5 for i in range(10)]  
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

```
>>> {i % 5 for i in range(5000)}  
set([0, 1, 2, 3, 4])
```

```
>>> import random  
>>> ".".join(str(random.randrange(256)) for i in  
range(4))
```

- Comprehensions are a handy way to build a list, set, dictionary, or tuple from another iterable type
- The join function on strings can be handy for turning a comprehension into a string

# Dictionary Comprehensions...

```
>>> {divisor:
```

```
    {numerator for numerator in range(20) if (numerator % divisor == 0)}  
    for divisor in range(1,20)  
    }
```

```
{1: set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]),  
2: set([0, 2, 4, 6, 8, 10, 12, 14, 16, 18]),  
3: set([0, 3, 6, 9, 12, 15, 18]),  
4: set([0, 4, 8, 12, 16]),    5: set([0, 5, 10, 15]),    6: set([0, 6, 12, 18]),  
7: set([0, 7, 14]),          8: set([0, 8, 16]),    9: set([0, 9, 18]),  
10: set([0, 10]),            11: set([0, 11]),    12: set([0, 12]),  
13: set([0, 13]),            14: set([0, 14]),    15: set([0, 15]),  
16: set([0, 16]),            17: set([0, 17]),    18: set([0, 18]),  
19: set([0, 19])}
```

# EXERCISE!

TODO: PUT AN EXERCISE IN HERE!!! Have folks solve some problem that's simplified using comprehensions



# Day 2

Screen I/O, File I/O, Socket I/O,  
exception handling  
Logging module

With and try are covered in here, and so  
are context managers (what "with" uses)

# Screen I/O

```
>>> print "hello world!"  
hello world!
```

```
>>> class foobar:  
...     def __str__(self):  
...         return "whatever man!"  
...     def __repr__(self):  
...         return "a representation of the parts"  
...
```

```
>>> print foobar  
__main__.foobar
```

```
>>> print foobar()  
whatever man!
```

```
>>> [foobar(), foobar()]  
[a representation of the parts, a representation  
of the parts]
```

- We've seen print, and we've seen future print...
- How does Python know how to print classes?
- Sometimes it's printing them nicely after conversion to string...
- Sometimes it's printing them as part of another data structure...
- You've undoubtedly seen these representations in error messages...

# Screen Formatting

```
>>> "An int {} a str {} and a list {} walk into a |".format(
    1, "1", [1,1,1]
)
'An int 1 a str 1 and a list [1, 1, 1] walk into a |'
```

```
>>> "An int %i a str %s and a list %s walk into a |" % \
    (1, "1",
     "[" + ",".join( str(i) for i in [1,1,1] ) + "]")
'An int 1 a str 1 and a list [1,1,1] walk into a |'
```

Recommendation:

- Use ".format" whenever possible
- Remember how to read % formatting...

Why? Some things you wish you could simply print aren't easy to print with %...

<https://docs.python.org/2/library/stdtypes.html#string-formatting-operations>

<https://docs.python.org/2/library/stdtypes.html#str.format>

<https://pyformat.info/>

# Screen I/O

```
>>> foobar = raw_input("Who is your daddy and what  
does he do?")
```

Who is your daddy and what does he do?

```
>>> He's a bad bad man!
```

```
>>> print foobar.split(" ")  
[, "He's", 'a', 'bad', 'bad', 'man!']
```

```
>>> print "Times two it's {}".format(  
    int(raw_input("Enter a number: ")) * 2  
)
```

Enter a number: 22

Times two it's 44

- Don't use input()!
  - Unless you're in python3, then:
  - ONLY USE input() !
- raw\_input returns a string...
- Why should you almost always avoid code like my "times two" example?

# File I/O

```
>>> with open("/tmp/terrible_secret","w") as fdes:
...     fdes.write("The terrible secret of space")
...
```

```
>>> with open("/tmp/terrible_secret","r") as fdes:
...     print fdes.read() + "<DONE>"
...
The terrible secret of space<DONE>
```

```
>>> fdes = open("/tmp/terrible_secret","a")
>>> fdes.write("\n")
>>> fdes.close()
```

```
>>> with open("/tmp/terrible_secret","r") as fdes:
...     print fdes.read() + "<DONE>"
...
The terrible secret of space
<DONE>
```

- ALRIGHT CORAL WHAT THE HECK IS THIS WITH THING!?!?!?
- "with" begins a context!
- Python guarantees that, upon exiting the context, it will properly close the context...
- In the case of a file descriptor given in the "as", it will close it
- Although - you don't have to use with...
- Write - doesn't add a newline...
- Read - by default reads the whole file...
- Check out the modes!
  - There's also the "b" mode addon...
  - <https://docs.python.org/2/tutorial/inputoutput.html#reading-and-writing-files>

# File I/O

```
>>> with open("/tmp/secrets","w") as f:
...     for line in ["kfc recipe","coke
ingredient","password"]:
...         f.write(line + "\n")
...
```

```
>>> type(f)
<type 'file'>
```

```
>>> with open("/tmp/secrets","r") as f:
...     for line in f:
...         print "The secret {}".format(line)
...
The secret kfc recipe
```

```
The secret coke ingredient
```

```
The secret password
```

- Check out what happens when you just iterate over the file descriptor variable!
  - Note the double newlines in the output...
- 
- Also - note that `f` is still defined after the block

# File I/O

```
>>> with open("/tmp/secrets","w") as f:  
...     for line in f:  
...         print "This is borked!"  
...
```

```
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
IOError: File not open for reading
```

```
>>> type(f)  
<type 'file'>
```

```
>>> f.write("asdf")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: I/O operation on closed file
```

- Here's an example of when "with" is handy
- Despite the error, the file closed!
- Note that f still exists after the "with"

# Other File Awesomeness

Tempfile - <https://docs.python.org/2/library/tempfile.html>

The os Package

os.path - <https://docs.python.org/2/library/os.path.html>

os.walk - <https://docs.python.org/2/library/os.html#os.walk>

os.listdir - <https://docs.python.org/2/library/os.html#os.listdir>

Pickle - <https://docs.python.org/2/library/pickle.html>

JSON - <https://docs.python.org/2/library/json.html>



# EXERCISE!!!

- Consider a simple tree data structure - a list of lists of lists of lists of...
  - The outermost list is the root
  - Branches are lists containing more branches and leaves, arbitrarily
  - Leaves are strings
- Input an arbitrary tree from a user, print it, save it to a file, and read it back again
- Consider just using json for the "save it to a file and read it" part...

Let's take 30 minutes for this one...

# EXERCISE!!!

What solutions did you guys come up with?

Maybe check out the reference solution...

# Socket I/O - TCP Client for IPV4 and IPV6, Simply

```
>>> import socket

>>> sock = socket.create_connection(("localhost",8080))

>>> sock.sendall("This is a test")

>>> sock.recv(4096)
'This is a response\n'

>>> sock.close()
```

- Making a TCP connection is too easy...
  - The socket module exposes lots of C-like functionality
  - Additionally, this "just handles" Windows situations
  - Use sendall to ensure all data is sent
  - Specify a maximum amount of data to recv
- 
- There's a select module too, if you want it, and it's very similar to C's

# Socket I/O - TCP and UDP Clients

```
>>> s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.connect(("localhost",8080))
>>> s.send("asdf")
4
>>> s.recv(4096)
'qwer\n'
>>> s.close()
```

```
>>> s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
>>> s.connect(("localhost",8080))
>>> s.sendall("asdf")
>>> s.recv(4096)
'qwer\n'
>>> s.close()
```

- Just like in C:
  - AF\_INET specifies IPV4
  - SOCK\_STREAM - TCP
  - SOCK\_DGRAM - UDP
  - Note - those are defined within "socket", so you must use "socket." before them...
- "send" is not guaranteed to send all data you gave it, so it returns the number of bytes actually sent

# Socket I/O - TCP Server

```
>>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
>>> sock.bind(("", 8080))
>>> sock.listen(10)
```

```
>>> client = sock.accept()
>>> print(client)
(<socket._socketobject object at 0x10b796ad0>, ('127.0.0.1', 51691))
```

```
>>> client[0].sendall("Here's some data")
>>> client[0].recv(4096)
"Here's a response\n"
>>> client[0].close()
```

```
>>> new_client = sock.accept()
>>> new_client[0].sendall("Here's some data")
>>> new_client[0].recv(4096)
'This is repetitive suckah!\n'
>>> new_client[0].close()
```

```
>>> sock.close()
```

- This looks a lot like C...
- Bind takes a tuple
  - First part specifies the address to bind to
  - Second part is port...
- Listen takes the number of connections to hold in backlog
  - The max value here is system dependent
  - It's usually maxed at 5
- Accept returns a tuple containing a tuple...
- Also - REUSEADDR...

# Socket I/O - UDP Server

```
>>> import socket
```

```
>>> sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
>>> sock.bind(("",8080))
```

```
>>> data, addr = sock.recvfrom(4096)
```

```
>>> data
```

```
"Here's some UDP test data\n"
```

```
>>> addr
```

```
('127.0.0.1', 61350)
```

```
>>> sock.sendto("Here's a response", addr)
```

```
17
```

```
>>> sock.close()
```

- Don't "listen"... `recvfrom`
- Watch out for "sendto" not sending all the data...

# When Things Go Wrong

```
>>> int("0x10")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '0x10'
```

```
>>> try:
...     int("0x10")
... except ValueError as e:
...     print("You did a bad bad thing")
... else:
...     print("You did a good good thing!")
... finally:
...     print("This always happens")
...
You did a bad bad thing
This always happens
```

- When Python hits something it can't handle, it raises an exception
- Many API functions specify which exceptions they'll throw
- In `_nearly all_` cases, catch only the exceptions you expect
  - Don't try to catch and handle them all
  - Unless - handling is to "print and die"
- If an exception is not caught, the function dies and raises the exception to the caller
- Except, else and finally should be at the same indentation as try

# When You Make Things Go Wrong

```
>>> def throw_stuff():
...     selection = raw_input("0, 1, or 2? ")
...     exceptions = [ValueError, RuntimeError,
...                   ReferenceError]
...     raise exceptions[int(selection)]("Message here...")

>>> def catch_stuff():
...     try:
...         throw_stuff()
...     except ValueError as e:
...         print "There was a value error {}".format(e)
...     except RuntimeError as e:
...         print "Runtime error this time {}".format(e)
...     except ReferenceError as e:
...         print "Reference error now fool! {}".format(e)
...     except Exception as e:
...         print "Some other exception dude {}".format(e)
...     except BaseException as e:
...         print "Oooh! We fell all the way back {}".format(e)
...     print "Still did this..."
```

Let's consider raising exceptions...

We'll define two functions:

- `throw_stuff` raises exceptions
  - Based on user input...
  - What if user input is not in the expected format
- `catch_stuff` calls `throw_stuff`
  - The programmer wanted to handle `throw_stuff`'s exceptions
  - Programmer caught all exceptions...
- Running this is on the next slide...



# When You Make Things Go Wrong

```
>>> catch_stuff()
```

```
0, 1, or 2? 0
```

```
There was a value error Message here...
```

```
Still did this...
```

```
>>> catch_stuff()
```

```
0, 1, or 2? 1
```

```
Runtime error this time Message here...
```

```
Still did this...
```

```
>>> catch_stuff()
```

```
0, 1, or 2? 2
```

```
Reference error now fool! Message here...
```

```
Still did this...
```

```
>>> catch_stuff()
```

```
0, 1, or 2? 3
```

```
Some other exception dude list index out of  
range
```

```
Still did this...
```

```
>>> catch_stuff()
```

```
0, 1, or 2? sdlfkj
```

```
There was a value error invalid literal for int()  
with base 10: 'sdlfkj'
```

```
Still did this...
```

- First - let's try expected inputs
- Then - try "3" - an invalid input
- Then - try "sdlfkj" - also invalid input, but one with an exception that was expected
  - If your code raises specific exceptions, try to use them for their expected purposes
  - Consider writing your own exception extending the "Exception" class
- What if the programmer had just written catch\_stuff wrong? Like - "throw\_stiff()" instead of "throw\_stuff()"?
- <https://docs.python.org/2/library/exceptions.html>

# Practical Exception Handling... Demo!!!

Sample\_tcp\_server.py is the tree exercise, over TCP!

However, leaves must be integers...

We'll step through the code quickly...

It is broken in a few places we'll have to debug...

There are several ways to break it just using common network comms...

Killing comms from the client's side...

Server's port is already taken...

Not giving an int as a leaf...

# EXERCISE!!!

Modify your tree program to get input and output over the network.

Handle the error cases you expect.

# Logging

```
>>> import logging
>>> logging.basicConfig()
>>> logging.debug("Debug output")
>>> logging.info("Info output")
>>> logging.warning("Warning output")
WARNING:root:Warning output
>>> logging.error("Error output")
ERROR:root:Error output
>>> logging.critical("Critical output")
CRITICAL:root:Critical output
```

- The logging module offers simplicity or complexity
  - `basicConfig` offers a variety of formatting options
  - Other modules that use logging will automatically use whatever format you specify
  - Want debug output and higher to a file, but info and higher to the screen? No problem.
  - Worried about output issues during threading? Don't be.
  - Want a simple way to annotate errors with the module they came in? Yeah, that's in there.
- 
- Check out the tutorial, it's awesome:
  - <https://docs.python.org/2/howto/logging.html>

# Unit Testing

```
""" to_test.py """
```

```
def multiply(one, two):  
    return one * two
```

```
def add(one, two):  
    return one + two
```

```
def subtract(one, two):  
    return one - two
```

```
#!/usr/bin/env python  
""" simple_tester.py """
```

```
import unittest
```

```
from to_test import add, multiply, subtract
```

```
class try_tests(unittest.TestCase):  
    def test_uno(self):  
        self.assertEqual(multiply(4,3), 12)  
        self.assertEqual(multiply(9,15), 135)  
        self.assertEqual(multiply("asdf", 4), "asdfasdfasdf")
```

```
    def test_dos(self):  
        self.assertAlmostEqual(multiply(5.0, 3.1394), 15.697, 6)  
        self.assertEqual(add(5.1, 3.1394), 8.2394)
```

```
if __name__ == "__main__":  
    unittest.main()
```

# Unit Testing

Step through more\_complex\_tester...

<https://docs.python.org/2/library/unittest.html>

# Day 3

Generators, iterators, context managers,  
lambda, functions-in-functions

THREADING - must hit, part of the  
exercise

Unit testing module should get hit by  
now...

# Generators and Iterators

Proceeding through lists, dictionary items, and strings with "for" is permitted because they are iterable

Iterable items require an `__iter__` function and a next function:

<https://docs.python.org/2/library/stdtypes.html#iterator-types>

There are three ways to implement this:

`separate_container_iterator_iterable.py`

`same_container_iterator_iterable.py`

`generator_iterable.py`



# Generators and Iterators - Case Study: Tuples

```
>>> tup = (1,2,3)
>>> type(tup)
<type 'tuple'>
>>> hasattr(tup, "__iter__")
True
>>> hasattr(tup, "next")
False
>>> tupiter = iter(tup)
>>> type(tupiter)
<type 'tupleiterator'>
>>> hasattr(tupiter, "__iter__")
True
>>> hasattr(tupiter, "next")
True
>>> [next(tupiter), next(tupiter), next(tupiter)]
[1, 2, 3]
>>> next(tupiter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- Tuples have an `__iter__`, but no `next`
- Passing `iter()` a tuple will return a `"tupleiterator"`
- The `tupleiterator` has both the `iter` and the `next`...
- Calling `next` on the `tupleiterator` will exhaust it eventually

# Generators and Iterators - Case Study: Strings

```
>>> str = "test str"
>>> hasattr(str, "__iter__")
False
>>> hasattr(str, "next")
False
>>> hasattr(str, "__getitem__")
True
>>> striter = iter(str)
>>> type(striter)
<type 'iterator'>
>>> hasattr(striter, "__iter__")
True
>>> hasattr(striter, "next")
True
>>> next(striter), next(striter), next(striter)
('t', 'e', 's')
```

- Strings, in python 2, don't have `__iter__` or `next`, yet, we've definitely iterated over them...
- They do have `"__getitem__"`, which takes an index and returns the character at that index
- Python's `"iter"` is able to use this to return something following the iterator protocol

# Day 4

Command line arguments

Itertools

HTTP Classes

Utilities bundled with python (2to3...)