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 antiquity. 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/
Last week

- Regular expressions
- Functional programming tools
- Generators
- File handing w/ the os module
This week

• Project guidelines
• Object oriented Python
• Exceptions
• Libraries part I
Course project
Project Proposal

• One page document describing
  – Problem statement / motivation
  – Expected input / output
  – Relevant libraries
  – Anticipated challenges / difficulties

• Timetable
  – Proposal due by the start of next class
  – Final project due by Tues March 2\textsuperscript{nd}
  – Live demo: must be scheduled via Doodle that same week (instructions to follow)

• Demo
  – 10 minute live demo, end to end run
  – If you need special hardware I can meet on campus

• See me if I can help you brainstorm ideas
Previous projects

• Genetic algorithms
• Solar system simulation in MAYA
• Music recommendations via mining Last.fm
• Financial engineering utilities
• Labview interface to monitor equipment
• Sports scheduling, game roster creation
• A webpage for elementary students
• Crypto
Requirements / Grading

• Originality
• Polish
• Technical
  – Design
  – Complexity
  – Execution
  – Library usage
• Documentation
• Effort
• Past questions
  – Line count
  – Interfaces
Object-oriented Python

Resources:
http://docs.python.org/tutorial/classes.html
Object oriented programming

• Object oriented paradigms
  – Classes
  – Instances
  – Inheritance
  – Polymorphism
  – Encapsulation
  – Operator overloading

• Python is a multi-paradigm language
  – You can mix and match procedural and OOP code
  – OOP is great when you need to group together data (state) and behavior (methods)
Class and Instances

• Classes
  – Classes define abstract objects which may be instantiated as instances. Classes are instance factories. Attributes provide data / state; methods provide functionality.

  A class is a user defined type

• Classes have attributes and methods
  – Class attributes are shared among instances
  – Instance attributes belong to specific instances

• Classes can be instantiated
  – Objects of that type are called instances

• Calling a class object returns an instances of that class

• Instances
  – Are instantiations of a class, represented by an object in a program.
Example

class Boat():
    def __init__(self, name): # the constructor
        self.name = name # an instance attribute
    def greet(self): # self refers to the calling instance
        print ('hi from', self.name)

betty = Boat('betty')
fred = Boat('fred')
betty.greet()
fred.greet()

$python foo.py
hi from betty
hi from fred
Attributes

• Attributes represent data which belong to the class or instances
• You can specify attributes inside the body
  – Descriptors (including functions), normal data objects, even other classes (nested classes)
• Attributes are specified by binding a value to an identifier inside or outside the body (binding inside is better for readability)
  – Can be bound at runtime
• The first string literal in the class body is taken to be the docstring
• Implicit attributes:
  – __name__: name of the class
  – __bases__: tuple of base classes
  – __dict__: dictionary object containing the class attributes
Class Attributes vs. Instance Attributes

class Boat():
    num_boats = 0
    def __init__(self, name):
        self.name = name
        Boat.num_boats +=1
    def greet(self):
        print ('hi from',
              self.name)

betty = Boat('betty')
betty.greet()
print (Boat.num_boats)
fred = Boat('fred')
fred.greet()
print (Boat.num_boats)

$python foo.py
hi from betty
1
hi from fred
2
Methods: class vs. instance

• Class methods are functions which typically run on data belonging to all classes
• Instance methods typically run on data belonging to a specific instance
• Methods can be defined in class bodies using the def statement
  – Instance method definitions have a mandatory first parameter: `self`
• `self` refers back to the instance which called the method, and is passed by Python automatically behind the scenes
• `class c(object):
  – def hello(self):
    • print (‘Hello world!’)
• Many types of methods can be defined (to be discussed later)
Self

- *self* is an *automatically* received first argument received when *instances* call methods
- *self* provides a reference back to the instance which called the class method
- Instance methods must specify *self* as their first parameter
- Class methods may be called without instantiating the class
- They do *not* use *self* as their first parameter
Example

class Homer():
    def eat():
        print ('Homer class method')
    def snack(self):
        print ('Homer instance method')

Homer.eat()
h = Homer()
h.snack() # self automatically passed

$python foo.py
Homer class method
Homer instance method
Classes and Instances (cont’d)

• Classes in Python are first-class objects
  – They are objects like any other
  – Can be passed as arguments to functions, used as keys in a dictionary, bound to local and global variables, etc.

• Classes work a lot like dictionaries: an instance of a class is a Python object with arbitrarily named attributes you can bind and reference

• Lookup of attributes not found in the instance itself is delegated to the class, which may be delegated to classes it inherits from
Constructors

• Constructor
  – If a class defines or inherits the __init__ method, it is implicitly executed when the class is instantiated
• To create an instance, call a class definition as if it were a function
  – myInstance = Foo()
• Calling a class object invokes the __init__ method on the new instance, deferring to the superclass if necessary
• __init__ bind’s attributes to the newly created instance
• Built-in function isinstance(I,C) returns True if object I is an instance of class C or its subclasses, False otherwise
The Class Statement

- class classname(base-classes):
  - statement(s)
- The class statement does not create any instances of the new classes, it simply defines their attributes and methods
  - __init__ is called only when an instance is created (and every time)
- base-classes are parents of the class, i.e. the current class derives or inherits from these base classes, is optional
- statement(s) is nonempty and is the class body, will execute immediately when the class statement is called
  - Until the body finishes executing, the class will not be bound to the identifier
- Caution: any executable code not in methods will run when the class definition is parsed
Inheritance

- Inheritance
  - Creating a new (sub) class by extending the functionality of an existing (parent or super) class. Results in the subclass inheriting the attributes and behavior of the parent class.
  - Inheritance in Python means that name lookup (for methods and attributes) is extended to the parent classes.
  - Python supports multiple inheritance
    - In case of conflicts between attributes or methods, the general rule is the first inherited class wins (left-most first).
Inheritance

class Boat():
    def __init__(self, name):
        self.name = name
    def greet(self):
        print ('hi from', self.name)

class Sailboat(Boat):
    def sail(self):
        print ('Wooosh')

b = Sailboat('betty') # the constructor is inherited
b.greet() # greet is inherited
b.sail() # sail is a new method specific to sailboats

$python foo.py
hi from betty
wooosh
Multiple Inheritance

class Sailboat():
    def sail(self):
        print ('Wooosh')

class Cannon():
    def fire(self):
        print ('Boom!')

class PirateShip(Sailboat, Cannon):
    pass

p = PirateShip()
p.sail()
p.fire()

$ python foo.py
Wooosh
Boom!
Method and Attribute Resolution

• Recall the syntax of the class statement
  
• class classname(base-classes):
  
    – statement(s)

• Python supports multiple inheritance
  
    – base-classes can be a comma-delimited list of superclasses

• Method resolution order
  
    – How does lookup of an attribute name occur?
    – In general: left-to-right, depth first
Composition

```python
class Homer():
    def __init__(self):
        self.donuts = []
    def add(self, donut):
        self.donuts.append(donut)
    def stats(self):
        print ('Homer has the following donuts:"
        for d in self.donuts:
            print ('\t' + d.name)

class Donut():
    def __init__(self, name):
        self.name = name

h = Homer()
h.add(Donut('jelly'))
h.add(Donut('sugar'))
h.stats()
```

$ python foo.py
Homer has the following donuts:
  jelly
  sugar
Polymorphism (overriding)

• Polymorphism
  – A subclass specializes the behavior of their parent class by overriding (or re-declaring) methods or data
  – Mammals swim (but people and dolphins swim rather differently)
• Polymorphism in python is as simple as re-declaring a method
• Common patterns:
  – Inheritor (does not override a method, makes use of the parent’s functionality)
  – Replacer (overrides the method entirely)
  – Extender (calls the parent’s method, but adds functionality)
  – Provider (fills in a template method declared by the parent)
Polymorphism example (replacer)

class Boat():
    def go(self):
        print ('Generic behavior')

class Sailboat(Boat):
    def go(self):
        print ('Let's go sailing!')

a = Boat()
a.go()
b = Sailboat()
b.go()

$ python foo.py
Generic behavior
Let's go sailing!
Overriding Attributes

• When a subclass defines an attribute with the same name as one in a superclass, the subclass’ attribute will always be used first
  – Known as the subclass *overriding the definition in the superclass*

• Delegating to superclass (or base) methods
  – Subclasses may call methods in base classes
Exposing functionality

• Python’s philosophy is to expose as much of a class as possible
• Private variables are signified by a leading underscore _
  – Decreases risk of accidental data sharing
  – But a convention that’s up to the programmers to respect
  – A determined programmer can access class private variables
class Boat():
    '''Class docs'''
    the_sky = 'blue'
    def go(self):
        print ('Generic behavior')

b = Boat()
print (b.__class__.__dict__)
Operator overloading

• Allows classes to define specific behavior for normal operators (e.g., +, -, *)
• As well as concepts such as iteration, type conversation, equality testing
• Useful if you’re developing a package
  – For instance, it makes sense to be able to multiple two vectors with the ‘*’ operator
• Use sparingly and only if obvious
Operator overloading example

class Donut():
    def __init__(self, name, quantity):
        self.name = name
        self.quantity = quantity
    def __add__(self, num):
        self.quantity += num
        print ('woohoo!')
        print ('we have %s donuts!' % (self.quantity))

d = Donut('jelly', 1)
d += 8

python foo.py
woohoo!
we have 9 donuts!
Providing iterator functionality

class Donut():
    def __init__(self, name):
        self.name = name
    def __getitem__(self, i):
        return self.name[i]

d = Donut('jelly')
print ('Give me a ', end = '')
for char in d:
    print (char + '!', end = ' ')

python foo.py
Give me a j! e! l! l! y!
Factory methods

• A factory is a function which returns an object of a particular class type depending on some condition

• A typical scenario is switching between two almost identical classes depending on the environment
class c1():
    def run_command(self):
        print ('ready for linux')

class c2():
    def run_command(self):
        print ('ready for windows')

def factory(linux=False):
    if linux:
        return c1()
    else:
        return c2()

x = factory(linux=True)
x.run_command()

$python foo.py
ready for linux
The object Type

• Built-in type: object
• Ancestor of all built-in types and new-style classes
• Some special methods are defined:
  – __new__, __init__, __delattr__, __del__, __hash__, __repr__, __str__, ...
Exceptions

Resources:
http://docs.python.org/tutorial/errors.html
Exceptions

• Difference between errors and exceptions?
  – Errors detected during execution are called *exceptions* and are not unconditionally fatal

• Python’s emphasis
  – Use exceptions when and where they make a program simpler, more robust, and more readable

• Special situations are frequently indicated in Python using exceptions
  – e.g., end of iteration is signaled by the StopIteration exception

• OK to use frequently
def bug():
    return 1 / 0

print (bug())

$ python f.py
Traceback (most recent call last):
  File "f.py", line 4, in <module>
    print (bug())
  File "f.py", line 2, in bug
    return 1 / 0
Exception objects

def bug():
    try:
        return 1 / 0
    except ZeroDivisionError as detail:
        print ('Caught a bug!')
        print (type(detail))
        print (detail)

print (bug())

$ python f.py
Caught a bug!
<class 'ZeroDivisionError'>
int division or modulo by zero
None
Stop Iteration

def count_down(to):
    while to > 0:
        to -= 1
        yield to

f = count_down(3)
while True:
    print(next(f))

$ python foo.py
2
1
0
Traceback (most recent call last):
  File "z.py", line 8, in <module>
    print(next(f))
StopIteration
Stop Iteration

def count_down(to):
    while to > 0:
        to -= 1
        yield to

f = count_down(3)
done = False
while not done:
    try:
        print(next(f))
    except StopIteration:
        print('all done')
        done = True
print('phew')

$ python foo.py
2
1
0
all done
phew
Exceptions

Raising Exceptions
• Exceptions communicate errors and anomalies
• When problems are detected, exceptions are raised / thrown
• Your code can explicitly raise exceptions
• Exceptions are caught by exception handlers
• Exceptions are instances of BaseException

Handling Exceptions
• Handling an exception means accepting the exception object from the propagation mechanism
• If exceptions are uncaught, they terminate the program and result in a stack trace
• Handling exceptions allows programs to deal with errors and anomalies gracefully
The try Statement

• Provides Python’s exception handling mechanism
• It is a compound statement with one of these forms:
  – Try clause followed by one or more except clauses (with optional else clause)
  – Try clause followed by finally clause
  – Try clause followed by except clauses and optional else clause, followed by finally clause (Python 2.5+)
Exception propagation

- When an exception is raised normal control flow is superseded by the exception propagation mechanism
- A raised exception is handled by the first try block with a matching except clause
- If an exception is raised without a try clause, or in a try clause without a matching except clause, it propagates up the call stack until either being caught, or terminating the program
- You can catch arbitrary deep exceptions produced by function calls
try/except/else

Syntax ([] indicate optional code):
try:
    statement(s)
except [expression [, target]]:
    statement(s)
[else:statement(s)]

• The body of the except clause is known as an exception handler
• Exception handler executes if expression matches an exception object propagating from the try clause
  – expression is an Exception class or tuple of classes
  – target is an identifier that is bound to the exception object before the handler executes
  – In the case of several except clauses, they are checked in order until one is found with a matching expression
  – List specific cases before general ones
try/except/else (cont’d)

• Last except may lack an expression
  – Known as bare excepts
  – Will handle any exception that reaches it
  – Should avoid; it’s sloppy coding
  – Trivia: “On error resume next”

• Exception propagation terminates when it finds a handler with a matching expression

• The optional else clause executes only when the try clause terminates normally (i.e. when no exception is raised) or when it exists with a break, continue or return statement
  – Handlers do not cover exceptions raised in the else clause
Examples

```python
>>> try: # try / except example
...     open('/')
... except IOError:
...     print ('Failed to open file.')
Failed to open file.

>>> try:
...     open('test', 'w')
...     print ('success')
... except IOError:
...     print ('Failed to create file')
... else:
...     print ('File creation succeeded.')
<open file 'test', mode 'w' at 0xb770f3e0>
Success
File creation succeeded.
```
Finally

Syntax
```python
try:
    statement(s)
finally:
    statement(s)
```

• The finally clause is a clean-up handler
  – It always executes after the try clause, regardless of whether or not an exception is raised (executes even if a return statement is placed w/in the try clause)
  – If an exception propagates from the try clause, the try clause will terminate, the finally clause executes, and the exception continues to propagate

• Specifies code which is guaranteed to run regardless of whether an exception occurs in the try block

• Useful to close database connections, files, etc
  – Wish the user a nice day before crashing
try/except/finally

- From Python 2.5 onward, except clause(s) are allowed with try/finally
- Syntax:
  ```python
try:
    statement(s)
except [expression[, target]]:
    statement(s)
finally:
    statement(s)
```
- Equivalent to:
  ```python
try:
    try:
      statement(s)
    except
      statement(s)
    finally:
      statement(s)
```
- If try clause raises an exception, it will be handled using the excepts before the finally clause is executed
- Can you think of some instances where try/except/finally would be useful?
The with statement

• New in Python 2.5 (standard in 2.6+, 3.x)
• Occasionally pops up in an error handling context
• Syntax:
  – with expression [as varname]
    • statement(s)
• Embodies the C++ idiom “resource acquisition is initialization”
• Best explained with an example:
  – with open(‘foo.txt’) as f:
    • statements using file object f
• More information:
Built-in exceptions

• (All of type Exception)
  – BaseException
  – AssertionError
  – AttributeError
  – IOError
  – ImportError
  – IndexError
  – KeyError
  – NotImplementedError
  – TypeError

• See: [http://docs.python.org/library/exceptions.html#bltin-exceptions](http://docs.python.org/library/exceptions.html#bltin-exceptions)
def homer_dates(x):
    assert(x != 'selma')
    print ('woohoo!')

homer_dates('marge')
woohoo!

homer_dates('selma')
Traceback (most recent call last):
  File "q.py", line 6, in <module>
    homer_dates('selma')
  File "q.py", line 2, in homer_dates
    assert(x != 'selma')
AssertionError
Defining your own exceptions

class HomerError(BaseException):
    '''Protects Homer'''

def homer_dates(x):
    if x == 'selma':
        raise HomerError
    print ('woohoo!')

try:
    homer_dates('marge')
    homer_dates('selma')
except HomerError:
    print ('not gonna happen')

$python foo.py
woohoo!
not gonna happen
Exception Handling Strategies

Look before you leap

Easier to ask forgiveness than permission
Python prefers the second

```python
def div(x, y):
    if y == 0:
        raise ZeroDivisionError
```

- Checks diminish readability
- Exceptions are rare, why waste effort up front?

```python
def div(x,y):
    try:
        return x / y
    except ZeroDivisionError:
        ...
```

- Emphasizes the common case
- Increases readability
Exceptions wrap up

- Avoid empty except statements
- Use the built-in exceptions before defining your own types
- Use assert as a sanity check
- The stack trace is powerful
- In small scripts, the easiest way to debug is often just to crash and examine it!
Libraries
XML-RPC

Anyone taken Networks? What do you think the line count would be in C?
XML-RPC:

http://docs.python.org/library/xmlrpclib.html

Server

```python
import xmlrpclib
from SimpleXMLRPCServer import SimpleXMLRPCServer

def is_even(n):
    return n%2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(is_even, "is_even")
server.serve_forever()
```

Client

```python
import xmlrpclib
proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
print proxy.is_even(3)
print proxy.is_even(100)

False
True
```
Finding and installing libraries

http://www.goldb.org/ystockquote.html

All it takes

```python
>>> import ystockquote

>>> ystockquote.get_price('GOOG')

357.95
```

Included Functions

- get_all(symbol)
- get_price(symbol)
- get_change(symbol)
- get_volume(symbol)
- get_avg_daily_volume(symbol)
- get_stock_exchange(symbol)
- get_market_cap(symbol)
- get_book_value(symbol)
- get_ebitda(symbol)
- get_dividend_per_share(symbol)
- ...