

PYTHON

PHILOSOPHY

From *The Zen of Python* (<https://www.python.org/dev/peps/pep-0020/>)

Beautiful is better than ugly.
 Explicit is better than implicit.
 Simple is better than complex.
 Complex is better than complicated.
 Flat is better than nested.
 Sparse is better than dense.
 Readability counts.
 Special cases aren't special enough to break the rules.
 Although practicality beats purity.
 Errors should never pass silently.
 Unless explicitly silenced.
 In the face of ambiguity, refuse the temptation to guess.
 There should be one-- and preferably only one --obvious way to do it.
 Although that way may not be obvious at first unless you're Dutch.
 Now is better than never.
 Although never is often better than *right now*.
 If the implementation is hard to explain, it's a bad idea.
 If the implementation is easy to explain, it may be a good idea.
 Namespaces are one honking great idea -- let's do more of those!

NOTABLE FEATURES

- Easy to learn.
- Supports quick development.
- Cross-platform.
- Open Source.
- Extensible.
- Embeddable.
- Large standard library and active community.
- Useful for a wide variety of applications.

INTERPRETER

- The standard implementation of Python is interpreted.
 - You can find info on various implementations [here](#).
- The interpreter translates Python code into bytecode, and this bytecode is executed by the Python VM (similar to Java).
- Two modes: normal and interactive.
 - Normal mode: entire .py files are provided to the interpreter.
 - Interactive mode: read-eval-print loop (REPL) executes statements piecewise.

INTERPRETER: NORMAL MODE

Let's write our first Python program!

In our favorite editor, let's create helloworld.py with the following contents:

```
print "Hello, World!"
```

From the terminal:

```
$ python helloworld.py
Hello, World!
```

Note: In Python 2.x, print is a statement. In Python 3.x, it is a function. If you want to get into the 3.x habit, include at the beginning:

```
from __future__ import print_function
```

Now, you can write

```
print("Hello, World!")
```

INTERPRETER: NORMAL MODE

Let's include a she-bang in the beginning of helloworld.py:

```
#!/usr/bin/env python
print "Hello, World!"
```

Now, from the terminal:

```
$ ./helloworld.py
Hello, World!
```

INTERPRETER: INTERACTIVE MODE

Let's accomplish the same task (and more) in interactive mode.

Some options:

- c : executes single command.
 - O: use basic optimizations.
 - d: debugging info.
- More can be found [here](#).

```
$ python
>>> print "Hello, World!"
Hello, World!
>>> hellostring = "Hello, World!"
>>> hellostring
'Hello, World!'
>>> 2*5
10
>>> 2*hellostring
'Hello, World!Hello, World!'
>>> for i in range(0,3):
...     print "Hello, World!"
...
Hello, World!
Hello, World!
Hello, World!
>>> exit()
$
```

SOME FUNDAMENTALS

- Whitespace is significant in Python. Where other languages may use {} or (), Python uses indentation to denote code blocks.

• Comments

- Single-line comments denoted by #.
- Multi-line comments begin and end with three "s.
- Typically, multi-line comments are meant for documentation.
- Comments should express information that cannot be expressed in code - do not restate code.

```
# here's a comment
for i in range(0,3):
    print i
def myfunc():
    """here's a comment about
    the myfunc function"""
    return "I'm in a function!"
```

PYTHON TYPING

- Python is a strongly, dynamically typed language.
- Strong Typing
 - Obviously, Python isn't performing static type checking, but it does prevent mixing operations between mismatched types.
 - Explicit conversions are required in order to mix types.
 - Example: `2 + "four"` ← not going to fly
- Dynamic Typing
 - All type checking is done at runtime.
 - No need to declare a variable or give it a type before use.

Let's start by looking at Python's built-in data types.

NUMERIC TYPES

The subtypes are int, long, float and complex.

- Their respective constructors are `int()`, `long()`, `float()`, and `complex()`.
- All numeric types, except complex, support the typical numeric operations you'd expect to find (a list is available [here](#)).
- Mixed arithmetic is supported, with the "narrower" type widened to that of the other. The same rule is used for mixed comparisons.

NUMERIC TYPES

- Numeric
 - **int**: equivalent to C's long int in 2.x but unlimited in 3.x.
 - **float**: equivalent to C's doubles.
 - **long**: unlimited in 2.x and unavailable in 3.x.
 - **complex**: complex numbers.
- Supported operations include constructors (i.e. `int(3)`), arithmetic, negation, modulus, absolute value, exponentiation, etc.

```
$ python
>>> 3 + 2
5
>>> 18 % 5
3
>>> abs(-7)
7
>>> float(9)
9.0
>>> int(5.3)
5
>>> complex(1,2)
(1+2j)
>>> 2 ** 8
256
```

SEQUENCE DATA TYPES

There are seven sequence subtypes: strings, Unicode strings, lists, tuples, bytearrays, buffers, and xrange objects.

All data types support arrays of objects but with varying limitations.

The most commonly used sequence data types are strings, lists, and tuples. The xrange data type finds common use in the construction of enumeration-controlled loops. The others are used less commonly.

SEQUENCE TYPES: STRINGS

Created by simply enclosing characters in either single- or double-quotes.

It's enough to simply assign the string to a variable.

Strings are immutable.

There are a tremendous amount of built-in string methods (listed [here](#)).

```
mystring = "Hi, I'm a string!"
```

SEQUENCE TYPES: STRINGS

Python supports a number of escape sequences such as '\t', '\n', etc.

Placing 'r' before a string will yield its raw value.

There is a string formatting operator '%' similar to C. A list of string formatting symbols is available [here](#).

Two string literals beside one another are automatically concatenated together.

```
print "\tHello,\n"
print r"\tWorld!\n"
print "Python is " "so cool."
```

```
$ python ex.py
Hello,
\tWorld!\n
Python is so cool.
```

SEQUENCE TYPES: UNICODE STRINGS

Unicode strings can be used to store and manipulate Unicode data.

As simple as creating a normal string (just put a 'u' on it!).

Use Unicode-Escape encoding for special characters.

Also has a raw mode, use 'ur' as a prefix.

To translate to a regular string, use the .encode() method.

To translate from a regular string to Unicode, use the unicode() function.

```
myunicodestr1 = u"Hi Class!"
myunicodestr2 = u"Hi\u0020Class!"
print myunicodestr1, myunicodestr2
newunicode = u'\xe4\xf6\xfc'
print newunicode
newstr = newunicode.encode('utf-8')
print newstr
print unicode(newstr, 'utf-8')
```

```
Output:
Hi Class! Hi Class!
äöü
äöü
äöü
```

SEQUENCE TYPES: LISTS

Lists are an incredibly useful *compound* data type.

Lists can be initialized by the constructor, or with a bracket structure containing 0 or more elements.

Lists are mutable – it is possible to change their contents. They contain the additional mutable operations.

Lists are nestable. Feel free to create lists of lists of lists...

```
mylist = [42, 'apple', u'unicode apple', 5234656]
print mylist
mylist[2] = 'banana'
print mylist
mylist[3] = [['item1', 'item2'], ['item3', 'item4']]
print mylist
mylist.sort()
print mylist
print mylist.pop()
mynewlist = [x+2 for x in range(0,5)]
print mynewlist
```

```
[42, 'apple', u'unicode apple', 5234656]
[42, 'apple', 'banana', 5234656]
[42, 'apple', 'banana', [['item1', 'item2'], ['item3', 'item4']]]
[42, [['item1', 'item2'], ['item3', 'item4']], 'apple', 'banana']
banana
[0, 2, 4, 6, 8]
```

SEQUENCE DATA TYPES

• Sequence

- **str**: string, represented as a sequence of 8-bit characters in Python 2.x.
- **unicode**: stores an abstract sequence of **code points**.
- **list**: a compound, mutable data type that can hold items of varying types.
- **tuple**: a compound, immutable data type that can hold items of varying types. Comma separated items surrounded by parentheses.
- a few more - we'll cover them later.

```
$ python
>>> mylist = ["spam", "eggs", "toast"] # List of strings!
>>> "eggs" in mylist
True
>>> len(mylist)
3
>>> mynewlist = ["coffee", "tea"]
>>> mylist + mynewlist
['spam', 'eggs', 'toast', 'coffee', 'tea']
>>> mytuple = tuple(mynewlist)
>>> mytuple
('coffee', 'tea')
>>> mytuple.index("tea")
1
>>> mylonglist = ['spam', 'eggs', 'toast', 'coffee', 'tea']
>>> mylonglist[2:4]
['toast', 'coffee']
```

COMMON SEQUENCE OPERATIONS

All sequence data types support the following operations.

Operation	Result
<code>x in s</code>	True if an item of <code>s</code> is equal to <code>x</code> , else False.
<code>x not in s</code>	False if an item of <code>s</code> is equal to <code>x</code> , else True.
<code>s + t</code>	The concatenation of <code>s</code> and <code>t</code> .
<code>s * n, n * s</code>	<code>n</code> shallow copies of <code>s</code> concatenated.
<code>s[i]</code>	<code>i</code> th item of <code>s</code> , origin 0.
<code>s[i:j]</code>	Slice of <code>s</code> from <code>i</code> to <code>j</code> .
<code>s[i:j:k]</code>	Slice of <code>s</code> from <code>i</code> to <code>j</code> with step <code>k</code> .
<code>len(s)</code>	Length of <code>s</code> .
<code>min(s)</code>	Smallest item of <code>s</code> .
<code>max(s)</code>	Largest item of <code>s</code> .
<code>s.index(x)</code>	Index of the first occurrence of <code>x</code> in <code>s</code> .

COMMON SEQUENCE OPERATIONS

Mutable sequence types further support the following operations.

Operation	Result
<code>s[i] = x</code>	Item <code>i</code> of <code>s</code> is replaced by <code>x</code> .
<code>s[i:j] = t</code>	Slice of <code>s</code> from <code>i</code> to <code>j</code> is replaced by the contents of <code>t</code> .
<code>del s[i:j]</code>	Same as <code>s[i:j] = []</code> .
<code>s[i:j:k] = t</code>	The elements of <code>s[i:j:k]</code> are replaced by those of <code>t</code> .
<code>del s[i:j:k]</code>	Removes the elements of <code>s[i:j:k]</code> from the list.
<code>s.append(x)</code>	Add <code>x</code> to the end of <code>s</code> .

COMMON SEQUENCE OPERATIONS

Mutable sequence types further support the following operations.

<code>s.extend(x)</code>	Appends the contents of <code>x</code> to <code>s</code> .
<code>s.count(x)</code>	Return number of <code>i</code> 's for which <code>s[i] == x</code> .
<code>s.index(x[, i[, j]])</code>	Return smallest <code>k</code> such that <code>s[k] == x</code> and <code>i <= k < j</code> .
<code>s.insert(i, x)</code>	Insert <code>x</code> at position <code>i</code> .
<code>s.pop([i])</code>	Same as <code>x = s[i]; del s[i]</code> ; return <code>x</code> .
<code>s.remove(x)</code>	Same as <code>del s[s.index(x)]</code> .
<code>s.reverse()</code>	Reverses the items of <code>s</code> in place.
<code>s.sort([cmp[, key[, reverse]])</code>	Sort the items of <code>s</code> in place.

BASIC BUILT-IN DATA TYPES

- Set
- **set**: an unordered collection of unique objects.
- **frozenset**: an immutable version of set.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange']
>>> fruit = set(basket)
>>> fruit
set(['orange', 'pear', 'apple'])
>>> 'orange' in fruit
True
>>> 'crabgrass' in fruit
False
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b
set(['r', 'd', 'b'])
>>> a | b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
```

BASIC BUILT-IN DATA TYPES

- Mapping
- **dict**: hash tables, maps a set of keys to arbitrary objects.

```
>>> gradebook = dict()
>>> gradebook['Susan Student'] = 87.0
>>> gradebook
{'Susan Student': 87.0}
>>> gradebook['Peter Pupil'] = 94.0
>>> gradebook.keys()
['Peter Pupil', 'Susan Student']
>>> gradebook.values()
[94.0, 87.0]
>>> gradebook.has_key('Tina Tenderfoot')
False
>>> gradebook['Tina Tenderfoot'] = 99.9
>>> gradebook
{'Peter Pupil': 94.0, 'Susan Student': 87.0, 'Tina Tenderfoot': 99.9}
>>> gradebook['Tina Tenderfoot'] = [99.9, 95.7]
>>> gradebook
{'Peter Pupil': 94.0, 'Susan Student': 87.0, 'Tina Tenderfoot': [99.9, 95.7]}
```

PYTHON DATA TYPES

So now we've seen some interesting Python data types.

Notably, we're very familiar with numeric types, strings, and lists.

That's not enough to create a useful program, so let's get some control flow tools under our belt.

CONTROL FLOW TOOLS

While loops have the following general structure.

```
while expression:
    statements
```

Here, *statements* refers to one or more lines of Python code. The conditional expression may be any expression, where any non-zero value is true. The loop iterates while the expression is true.

Note: All the statements indented by the same amount after a programming construct are considered to be part of a single block of code.

```
i = 1
while i < 4:
    print i
    i = i + 1
flag = True
while flag and i < 8:
    print flag, i
    i = i + 1
```

```
1
2
3
True 4
True 5
True 6
True 7
```

CONTROL FLOW TOOLS

The if statement has the following general form.

```
if expression:
    statements
```

If the boolean expression evaluates to True, the statements are executed. Otherwise, they are skipped entirely.

```
a = 1
b = 0
if a:
    print "a is true!"
if not b:
    print "b is false!"
if a and b:
    print "a and b are true!"
if a or b:
    print "a or b is true!"
```

a is true!
b is false!
a or b is true!

CONTROL FLOW TOOLS

You can also pair an else with an if statement.

```
if expression:
    statements
else:
    statements
```

The elif keyword can be used to specify an else if statement.

Furthermore, if statements may be nested within each other.

```
a = 1
b = 0
c = 2
if a > b:
    if a > c:
        print "a is greatest"
    else:
        print "c is greatest"
elif b > c:
    print "b is greatest"
else:
    print "c is greatest"
```

c is greatest

CONTROL FLOW TOOLS

The for loop has the following general form.

```
for var in sequence:
    statements
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *var*. Next, the statements are executed. Each item in the sequence is assigned to *var*, and the statements are executed until the entire sequence is exhausted.

For loops may be nested with other control flow tools such as while loops and if statements.

```
for letter in "aeiou":
    print "vowel: ", letter
for i in [1,2,3]:
    print i
for i in range(0,3):
    print i
```

```
vowel: a
vowel: e
vowel: i
vowel: o
vowel: u
1
2
3
0
1
2
```

CONTROL FLOW TOOLS

Python has two handy functions for creating a range of integers, typically used in for loops. These functions are `range()` and `xrange()`.

They both create a sequence of integers, but `range()` creates a list while `xrange()` creates an xrange object.

Essentially, `range()` creates the list statically while `xrange()` will generate items in the list as they are needed. We will explore this concept further in just a week or two.

For very large ranges – say one billion values – you should use `xrange()` instead. For small ranges, it doesn't matter.

```
for i in xrange(0, 4):
    print i
for i in range(0,8,2):
    print i
for i in range(20,14,-2):
    print i
```

```
0
1
2
3
0
2
4
6
20
18
16
```

CONTROL FLOW TOOLS

There are four statements provided for manipulating loop structures. These are break, continue, pass, and else.

- break: terminates the current loop.
- continue: immediately begin the next iteration of the loop.
- pass: do nothing. Use when a statement is required syntactically.
- else: represents a set of statements that should execute when a loop terminates.

```
for num in range(10,20):
    if num%2 == 0:
        continue
    for i in range(3,num):
        if num%i == 0:
            break
    else:
        print num, 'is a prime number'
```

```
11 is a prime number
13 is a prime number
17 is a prime number
19 is a prime number
```

OUR FIRST REAL PYTHON PROGRAM

Ok, so we got some basics out of the way. Now, we can try to create a real program.

I pulled a problem off of [Project Euler](#). Let's have some fun.

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

A SOLUTION USING BASIC PYTHON

```
from __future__ import print_function

total = 0
f1, f2 = 1, 2
while f1 < 4000000:
    if f1 % 2 == 0:
        total = total + f1
    f1, f2 = f2, f1 + f2
print(total)
```

Output: 4613732

Notice we're using the Python 3.x version of print here.

Python supports multiple assignment at once. Right hand side is fully evaluated before setting the variables.

FUNCTIONS

A function is created with the def keyword. The statements in the block of the function must be indented.

```
def function_name(args):
    statements
```

The def keyword is followed by the function name with round brackets enclosing the arguments and a colon. The indented statements form a body of the function.

The return keyword is used to specify a list of values to be returned.

```
# Defining the function
def print_greeting():
    print "Hello!"
    print "How are you today?"
```

```
print_greeting() # Calling the function
```

```
Hello!
How are you today?
```


FUNCTIONS

All parameters in the Python language are passed by reference.

However, only mutable objects can be changed in the called function.

We will talk about this in more detail later.

```
def hello_func(name, somelist):
    print "Hello,", name, "!\n"
    name = "Caitlin"
    somelist[0] = 3
    return 1, 2

myname = "Ben"
mylist = [1,2]
a,b = hello_func(myname, mylist)
print myname, mylist
print a, b
```

```
Hello, Ben !

Ben [3, 2]
1 2
```

FUNCTIONS

What is the output of the following code?

```
def hello_func(names):
    for n in names:
        print "Hello,", n, "!"
    names[0] = 'Susie'
    names[1] = 'Pete'
    names[2] = 'Will'
    names = ['Susan', 'Peter', 'William']
    hello_func(names)
    print "The names are now", names, "."
```

```
Hello, Susan !
Hello, Peter !
Hello, William !
The names are now ['Susie', 'Pete', 'Wi
```

A SOLUTION WITH FUNCTIONS

The Python interpreter will set some special environmental variables when it starts executing.

If the Python interpreter is running the module (the source file) as the main program, it sets the special `__name__` variable to have a value `"__main__"`. This allows for flexibility in writing your modules.

Note: `__name__`, as with other built-ins, has two underscores on either side!

```
from __future__ import print_function

def even_fib():
    total = 0
    f1, f2 = 1, 2
    while f1 < 4000000:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    print(even_fib())
```

INPUT

• raw_input()

- Asks the user for a string of input, and returns the string.
- If you provide an argument, it will be used as a prompt.

```
>>> print(raw_input('What is your name? '))
What is your name? Caitlin
Caitlin
>>> print(input('Do some math: '))
Do some math: 2+2*5
12
```

• input()

- Uses `raw_input()` to grab a string of data, but then tries to evaluate the string as if it were a Python expression.

- Returns the value of the expression.
- Dangerous – don't use it.

Note: In Python 3.x, `input()` is now just an alias for `raw_input()`

A SOLUTION WITH INPUT

```
from __future__ import print_function
```

```
def even_fib(n):
```

```
    total = 0
```

```
    f1, f2 = 1, 2
```

```
    while f1 < n:
```

```
        if f1 % 2 == 0:
```

```
            total = total + f1
```

```
        f1, f2 = f2, f1 + f2
```

```
    return total
```

```
if __name__ == "__main__":
```

```
    limit = raw_input("Enter the max Fibonacci number: ")
```

```
    print(even_fib(int(limit)))
```

```
Enter the max Fibonacci number: 4000000
4613732
```

CODING STYLE

So now that we know how to write a Python program, let's break for a bit to think about our coding style. Python has a style guide that is useful to follow, you can read about PEP 8 [here](#).

I encourage you all to check out [pylint](#), a Python source code analyzer that helps you maintain good coding standards.

MODULES

```
''' Module fib.py '''
from __future__ import print_function
```

```
def even_fib(n):
```

```
    total = 0
```

```
    f1, f2 = 1, 2
```

```
    while f1 < n:
```

```
        if f1 % 2 == 0:
```

```
            total = total + f1
```

```
        f1, f2 = f2, f1 + f2
```

```
    return total
```

```
if __name__ == "__main__":
```

```
    limit = raw_input("Max Fibonacci number: ")
```

```
    print(even_fib(int(limit)))
```

So, we just put together our first real Python program. Let's say we store this program in a file called fib.py.

We have just created a *module*.

Modules are simply text files containing Python definitions and statements which can be executed directly or imported by other modules.

MODULES

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix .py appended.
- Within a module, the module's name (as a string) is available as the value of the global variable `__name__`.
- If a module is executed directly however, the value of the global variable `__name__` will be `"__main__"`.
- Modules can contain executable statements aside from definitions. These are executed only the *first* time the module name is encountered in an import statement as well as if the file is executed as a script.

MODULES

I can run our module directly at the command line. In this case, the module's `__name__` variable has the value `"__main__"`.

```
$ python fib.py
Max Fibonacci number: 4000000
4613732
```

```
''' Module fib.py '''
from __future__ import print_function

def even_fib(n):
    total = 0
    f1, f2 = 1, 2
    while f1 < n:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    limit = raw_input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MODULES

I can import the module into the interpreter. In this case, the value of `__name__` is simply the name of the module itself.

```
$ python
>>> import fib
>>> fib.even_fib(4000000)
4613732
```

```
''' Module fib.py '''
from __future__ import print_function

def even_fib(n):
    total = 0
    f1, f2 = 1, 2
    while f1 < n:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    limit = raw_input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MODULES

I can import the module into the interpreter. In this case, the value of `__name__` is simply the name of the module itself.

```
$ python
>>> import fib
>>> fib.even_fib(4000000)
4613732
```

Note that we can only access the definitions of `fib` as members of the `fib` object.

```
''' Module fib.py '''
from __future__ import print_function

def even_fib(n):
    total = 0
    f1, f2 = 1, 2
    while f1 < n:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    limit = raw_input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MODULES

I can import the definitions of the module directly into the interpreter.

```
$ python
>>> from fib import even_fib
>>> even_fib(4000000)
4613732
```

To import *everything* from a module:

```
>>> from fib import *
```

```
''' Module fib.py '''
from __future__ import print_function

def even_fib(n):
    total = 0
    f1, f2 = 1, 2
    while f1 < n:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    limit = raw_input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

FUNCTIONS

We already know the basics of functions so let's dive a little deeper.

Let's say we write a function in Python which allows a user to connect to a remote machine using a username/password combination. Its signature might look something like this:

```
def connect(uname, pword, server, port):
    print "Connecting to", server, ":", port, "..."
    # Connecting code here ...
```

We've created a function called *connect* which accepts a username, password, server address, and port as arguments (in that order!).

FUNCTIONS

```
def connect(uname, pword, server, port):
    print "Connecting to", server, ":", port, "..."
    # Connecting code here ...
```

Here are some example ways we might call this function:

- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu', 9160)`
- `connect('jdoe', 'r5f0g87g5@y', 'linprog.cs.fsu.edu', 6370)`

FUNCTIONS

These calls can become a little cumbersome, especially if one of the arguments is likely to have the same value for every call.

Default argument values

- We can provide a default value for any number of arguments in a function.
- Allows functions to be called with a variable number of arguments.
- Arguments with default values must appear at the end of the arguments list!

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

FUNCTIONS

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

Now we can provide a variable number of arguments. All of the following calls are valid:

- `connect('admin', 'ilovecats')`
- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu')`
- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu', 6379)`

SURPRISING BEHAVIOR

Let's say I have the following Python module. It defines the `add_item` function whose arguments are `item` and `item_list`, which defaults to an empty list.

```
''' Module adder.py '''

def add_item(item, item_list = []):
    item_list.append(item) # Add item to end of list
    print item_list
```

SURPRISING BEHAVIOR

Let's say I have the following Python module. It defines the `add_item` function whose arguments are `item` and `item_list`, which defaults to an empty list.

```
''' Module adder.py '''

def add_item(item, item_list = []):
    item_list.append(item)
    print item_list

$ python
>>> from adder import *
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[4, 5]
```

SURPRISING BEHAVIOR

This bizarre behavior actually gives us some insight into how Python works.

```
''' Module adder.py '''

def add_item(item, item_list = []):
    item_list.append(item)
    print item_list

$ python
>>> from adder import *
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[4, 5]
```

Python's default arguments are evaluated *once* when the function is defined, not every time the function is called. This means that if you make changes to a mutable default argument, these changes will be reflected in future calls to the function.

SURPRISING BEHAVIOR

This bizarre behavior actually gives us some insight into how Python works.

```
''' Module adder.py '''

def add_item(item, item_list = []):
    item_list.append(item)
    print item_list

$ python
>>> from adder import *
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[4, 5]
```

Arguments are evaluated at this point!

Python's default arguments are evaluated *once* when the function is defined, not every time the function is called. This means that if you make changes to a mutable default argument, these changes will be reflected in future calls to the function.

SURPRISING BEHAVIOR

An easy fix is to use a sentinel default value that tells you when to create a new mutable argument.

```
''' Module adder.py '''

def add_item(item, item_list = None):
    if item_list == None:
        item_list = []
    item_list.append(item)
    print item_list

$ python
>>> from adder import *
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[5]
```

FUNCTIONS

Consider again our connecting function.

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

The following call utilizes *positional arguments*. That is, Python determines which formal parameter to bind the argument to based on its position in the list.

```
connect('admin', 'ilovecats', 'shell.cs.fsu.edu', 6379)
```

FUNCTIONS

When the formal parameter is specified, this is known as a *keyword argument*.

```
connect(uname='admin', pword='ilovecats',
        server='shell.cs.fsu.edu', port=6379)
```

By using keyword arguments, we can explicitly tell Python to which formal parameter the argument should be bound. Keyword arguments are always of the form *kwarg = value*.

If keyword arguments are used they must follow any positional arguments, although the relative order of keyword arguments is unimportant.

FUNCTIONS

Given the following function signature, which of the following calls are valid?

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

1. `connect('admin', 'ilovecats', 'shell.cs.fsu.edu')`
2. `connect(uname='admin', pword='ilovecats', 'shell.cs.fsu.edu')`
3. `connect('admin', 'ilovecats', port=6379, server='shell.cs.fsu.edu')`

FUNCTIONS

Given the following function signature, which of the following calls are valid?

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

1. `connect('admin', 'ilovecats', 'shell.cs.fsu.edu')` -- VALID
2. `connect(uname='admin', pword='ilovecats', 'shell.cs.fsu.edu')` -- INVALID
3. `connect('admin', 'ilovecats', port=6379, server='shell.cs.fsu.edu')` -- VALID

FUNCTIONS

Formal parameters of the form `*param` contain a variable number of arguments within a tuple. Formal parameters of the form `**param` contain a variable number of keyword arguments.

```
def connect(uname, *args, **kwargs):
    # connecting code here
```

This is known as *packing*.

Within the function, we can treat `args` as a list of the positional arguments provided and `kwargs` as a dictionary of keyword arguments provided.

FUNCTIONS

```
def example(param1, *args, **kwargs):
    print "param1: ", param1
    for arg in args:
        print arg
    for key in kwargs.keys():
        print key, ":", kwargs[key]
```

```
example('one', 'two', 'three', server='localhost', port=9160)
```

Output: ?

FUNCTIONS

We can use `*args` and `**kwargs` not only to define a function, but also to call a function. Let's say we have the following function.

```
def func(arg1, arg2, arg3):
    print "arg1:", arg1
    print "arg2:", arg2
    print "arg3:", arg3
```

FUNCTIONS

We can use `*args` to pass in a tuple as a single argument to our function. This tuple should contain the arguments in the order in which they are meant to be bound to the formal parameters.

```
>>> args = ("one", 2, 3)
>>> func(*args)
arg1: one
arg2: 2
arg3: 3
```

We would say that we're *unpacking* a tuple of arguments here.

FUNCTIONS

We can use `**kwargs` to pass in a dictionary as a single argument to our function. This dictionary contains the formal parameters as keywords, associated with their argument values. Note that these can appear in any order.

```
>>> kwargs = {"arg3": 3, "arg1": "one", "arg2": 2}
>>> func(**kwargs)
arg1: one
arg2: 2
arg3: 3
```

LAMBDA FUNCTIONS

One can also define lambda functions within Python.

- Use the keyword *lambda* instead of *def*.
- Can be used wherever function objects are used.
- Restricted to one expression.
- Typically used with functional programming tools – we will see this next time.

```
>>> def f(x):
...     return x**2
...
>>> print f(8)
64
>>> g = lambda x: x**2
>>> print g(8)
64
```

LIST COMPREHENSIONS

List comprehensions provide a nice way to construct lists where the items are the result of some operation.

The simplest form of a list comprehension is

```
[expr for x in sequence]
```

Any number of additional `for` and/or `if` statements can follow the initial `for` statement. A simple example of creating a list of squares:

```
>>> squares = [x**2 for x in range(0,11)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```


LIST COMPREHENSIONS

Here's a more complicated example which creates a list of tuples.

```
>>> squares = [(x, x**2, x**3) for x in range(0,9) if x % 2 == 0]
>>> squares
[(0, 0, 0), (2, 4, 8), (4, 16, 64), (6, 36, 216), (8, 64, 512)]
```

The initial expression in the list comprehension can be anything, even another list comprehension.

```
>>> [[x*y for x in range(1,5)] for y in range(1,5)]
[[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12], [4, 8, 12, 16]]
```

MORE DATA STRUCTURES

- Lists
 - Slicing
 - Stacks and Queues
- Tuples
- Sets and Frozensets
- Dictionaries
- How to choose a data structure.
- Collections
 - Deques and OrderedDicts

WHEN TO USE LISTS

- When you need a collection of elements of varying type.
- When you need the ability to order your elements.
- When you need the ability to modify or add to the collection.
- When you don't require elements to be indexed by a custom value.
- When you need a stack or a queue.
- When your elements are not necessarily unique.

CREATING LISTS

To create a list in Python, we can use bracket notation to either create an empty list or an initialized list.

```
mylist1 = [] # Creates an empty list
mylist2 = [expression1, expression2, ...]
mylist3 = [expression for variable in sequence]
```

The first two are referred to as *list displays*, where the last example is a *list comprehension*.

CREATING LISTS

We can also use the built-in list constructor to create a new list.

```
mylist1 = list()
mylist2 = list(sequence)
mylist3 = list(expression for variable in sequence)
```

The sequence argument in the second example can be any kind of sequence object or iterable. If another list is passed in, this will create a copy of the argument list.

CREATING LISTS

Note that you cannot create a new list through assignment.

```
# mylist1 and mylist2 point to the same list
mylist1 = mylist2 = []

# mylist3 and mylist4 point to the same list
mylist3 = []
mylist4 = mylist3

mylist5 = []; mylist6 = [] # different lists
```

ACCESSING LIST ELEMENTS

If the index of the desired element is known, you can simply use bracket notation to index into the list.

```
>>> mylist = [34,67,45,29]
>>> mylist[2]
45
```

If the index is not known, use the `index()` method to find the first index of an item. An exception will be raised if the item cannot be found.

```
>>> mylist = [34,67,45,29]
>>> mylist.index(67)
1
```

SLICING AND SLIDING

- The length of the list is accessible through `len(mylist)`.
- Slicing is an extended version of the indexing operator and can be used to grab sublists.

```
mylist[start:end] # items start to end-1
mylist[start:]    # items start to end of the array
mylist[:end]      # items from beginning to end-1
mylist[:]         # a copy of the whole array
```

- You may also provide a step argument with any of the slicing constructions above.

```
mylist[start:end:step] # start to end-1, by step
```

SLICING AND SLIDING

- The start or end arguments may be negative numbers, indicating a count from the end of the array rather than the beginning. This applies to the indexing operator.

```
mylist[-1]      # last item in the array
mylist[-2:]     # last two items in the array
mylist[:-2]     # everything except the last two items
```

- Some examples:

```
mylist = [34, 56, 29, 73, 19, 62]
mylist[-2]      # yields 19
mylist[-4::2]   # yields [29, 19]
```

INSERTING/REMOVING ELEMENTS

- To add an element to an existing list, use the `append()` method.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.append(47)
>>> mylist
[34, 56, 29, 73, 19, 62, 47]
```

- Use the `extend()` method to add all of the items from another list.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.extend([47, 81])
>>> mylist
[34, 56, 29, 73, 19, 62, 47, 81]
```

INSERTING/REMOVING ELEMENTS

- Use the `insert(pos, item)` method to insert an item at the given position. You may also use negative indexing to indicate the position.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.insert(2, 47)
>>> mylist
[34, 56, 47, 29, 73, 19, 62]
```

- Use the `remove()` method to remove the first occurrence of a given item. An exception will be raised if there is no matching item in the list.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.remove(29)
>>> mylist
[34, 56, 73, 19, 62]
```

LISTS AS STACKS

- You can use lists as a quick stack data structure.
- The `append()` and `pop()` methods implement a LIFO structure.
- The `pop(index)` method will remove and return the item at the specified index. If no index is specified, the last item is popped from the list.

```
>>> stack = [34, 56, 29, 73, 19, 62]
>>> stack.append(47)
>>> stack
[34, 56, 29, 73, 19, 62, 47]
>>> stack.pop()
47
>>> stack
[34, 56, 29, 73, 19, 62]
```

LISTS AS QUEUES

- Lists *can* be used as queues natively since `insert()` and `pop()` both support indexing. However, while appending and popping from a list are fast, inserting and popping from the beginning of the list are slow.

- Use the special *deque* object from the *collections* module.

```
>>> from collections import deque
>>> queue = deque([35, 19, 67])
>>> queue.append(42)
>>> queue.append(23)
>>> queue.popleft()
35
>>> queue.popleft()
19
>>> queue
deque([67, 42, 23])
```

OTHER OPERATIONS

- The `count(x)` method will give you the number of occurrences of item `x` within the list.

```
>>> mylist = ['a', 'b', 'c', 'd', 'a', 'f', 'c']
>>> mylist.count('a')
2
```

- The `sort()` and `reverse()` methods sort and reverse the list in place. The `sorted(mylist)` and `reversed(mylist)` built-in functions will return a sorted and reversed copy of the list, respectively.

```
>>> mylist = [5, 2, 3, 4, 1]
>>> mylist.sort()
>>> mylist
[1, 2, 3, 4, 5]
>>> mylist.reverse()
>>> mylist
[5, 4, 3, 2, 1]
```

CUSTOM SORTING

- Both the `sorted()` built-in function and the `sort()` method of lists accept some optional arguments.

```
sorted(iterable[, cmp[, key[, reverse]]])
```

- The `cmp` argument specifies a custom comparison function of two arguments which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument. The default value is `None`.
- The `key` argument specifies a function of one argument that is used to extract a comparison key from each list element. The default value is `None`.
- The `reverse` argument is a Boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

CUSTOM SORTING

```
>>> mylist = ['b', 'A', 'D', 'c']
>>> mylist.sort(cmp = lambda x,y: cmp(x.lower(), y.lower()))
>>> mylist
['A', 'b', 'c', 'D']
```

Alternatively,

```
>>> mylist = ['b', 'A', 'D', 'c']
>>> mylist.sort(key = str.lower)
>>> mylist
['A', 'b', 'c', 'D']
```

`str.lower()` is a built-in string method.

WHEN TO USE SETS

- When the elements must be unique.
- When you need to be able to modify or add to the collection.
- When you need support for mathematical set operations.
- When you don't need to store nested lists, sets, or dictionaries as elements.

CREATING SETS

- Create an empty set with the set constructor.

```
myset = set()
myset2 = set([]) # both are empty sets
```

- Create an initialized set with the set constructor or the {} notation. Do not use empty curly braces to create an empty set – you'll get an empty dictionary instead.

```
myset = set(sequence)
myset2 = {expression for variable in sequence}
```

HASHABLE ITEMS

The way a set detects non-unique elements is by indexing the data in memory, creating a hash for each element. This means that all elements in a set must be *hashable*.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are also hashable by default.

MUTABLE OPERATIONS

The following operations are not available for frozensets.

- The `add(x)` method will add element `x` to the set if it's not already there. The `remove(x)` and `discard(x)` methods will remove `x` from the set.
- The `pop()` method will remove and return an arbitrary element from the set. Raises an error if the set is empty.
- The `clear()` method removes all elements from the set.

```
>>> myset = set('abracadabra')
>>> myset
set(['a', 'b', 'r', 'c', 'd'])
>>> myset.add('y')
>>> myset
set(['a', 'b', 'r', 'c', 'd', 'y'])
>>> myset.remove('a')
>>> myset
set(['b', 'r', 'c', 'd', 'y'])
>>> myset.pop()
'b'
>>> myset
set(['r', 'c', 'd', 'y'])
```

MUTABLE OPERATIONS CONTINUED

```
set |= other | ...
```

Update the set, adding elements from all others.

```
set &= other & ...
```

Update the set, keeping only elements found in it and all others.

```
set -= other | ...
```

Update the set, removing elements found in others.

```
set ^= other
```

Update the set, keeping only elements found in either set, but not in both.

MUTABLE OPERATIONS CONTINUED

```
>>> s1 = set('abracadabra')
>>> s2 = set('alacazam')
>>> s1
set(['a', 'b', 'r', 'c', 'd'])
>>> s2
set(['a', 'l', 'c', 'z', 'm'])
>>> s1 |= s2
>>> s1
set(['a', 'b', 'r', 'c', 'd', 'l', 'z', 'm'])
>>> s1 = set('abracadabra')
>>> s1 &= s2
>>> s1
set(['a', 'c'])
```

SET OPERATIONS

- The following operations are available for both set and frozenset types.
- Comparison operators `>=`, `<=` test whether a set is a superset or subset, respectively, of some other set. The `>` and `<` operators check for proper supersets/subsets.

```
>>> s1 = set('abracadabra')
>>> s2 = set('bard')
>>> s1 >= s2
True
>>> s1 > s2
True
>>> s1 <= s2
False
```

SET OPERATIONS

- Union: `set | other | ...`
 - Return a new set with elements from the set and all others.
- Intersection: `set & other & ...`
 - Return a new set with elements common to the set and all others.
- Difference: `set - other - ...`
 - Return a new set with elements in the set that are not in the others.
- Symmetric Difference: `set ^ other`
 - Return a new set with elements in either the set or other but not both.

SET OPERATIONS

```
>>> s1 = set('abracadabra')
>>> s1
set(['a', 'b', 'r', 'c', 'd'])
>>> s2 = set('alacazam')
>>> s2
set(['a', 'l', 'c', 'z', 'm'])
>>> s1 | s2
set(['a', 'b', 'r', 'c', 'd', 'l', 'z', 'm'])
>>> s1 & s2
set(['a', 'c'])
>>> s1 - s2
set(['b', 'r', 'd'])
>>> s1 ^ s2
set(['b', 'r', 'd', 'l', 'z', 'm'])
```

OTHER OPERATIONS

- `s.copy()` returns a shallow copy of the set `s`.
- `s.isdisjoint(other)` returns True if set `s` has no elements in common with set `other`.
- `s.issubset(other)` returns True if set `s` is a subset of set `other`.
- `len`, `in`, and `not in` are also supported.

WHEN TO USE TUPLES

- When storing elements that will not need to be changed.
- When performance is a concern.
- When you want to store your data in logical immutable pairs, triples, etc.

CONSTRUCTING TUPLES

- An empty tuple can be created with an empty set of parentheses.
- Pass a sequence type object into the `tuple()` constructor.
- Tuples can be initialized by listing comma-separated values. These do not need to be in parentheses but they can be.
- One quirk: to initialize a tuple with a single value, use a trailing comma.

```
>>> t1 = (1, 2, 3, 4)
>>> t2 = "a", "b", "c", "d"
>>> t3 = ()
>>> t4 = ("red", )
```

TUPLE OPERATIONS

Tuples are very similar to lists and support a lot of the same operations.

- Accessing elements: use bracket notation (e.g. `t1[2]`) and slicing.
- Use `len(t1)` to obtain the length of a tuple.
- The universal immutable sequence type operations are all supported by tuples.
 - `+`, `*`
 - `in`, `not in`
 - `min(t)`, `max(t)`, `t.index(x)`, `t.count(x)`

PACKING/UNPACKING

Tuple packing is used to “pack” a collection of items into a tuple. We can unpack a tuple using Python’s multiple assignment feature.

```
>>> s = "Susan", 19, "CS" # tuple packing
>>> name, age, major = s # tuple unpacking
>>> name
'Susan'
>>> age
19
>>> major
'CS'
```

WHEN TO USE DICTIONARIES

- When you need to create associations in the form of key:value pairs.
- When you need fast lookup for your data, based on a custom key.
- When you need to modify or add to your key:value pairs.

CONSTRUCTING A DICTIONARY

- Create an empty dictionary with empty curly braces or the `dict()` constructor.
- You can initialize a dictionary by specifying each key:value pair within the curly braces.
- Note that keys must be *hashable* objects.

```
>>> d1 = {}
>>> d2 = dict() # both empty
>>> d3 = {"Name": "Susan", "Age": 19, "Major": "CS"}
>>> d4 = dict(Name="Susan", Age=19, Major="CS")
>>> d5 = dict(zip(['Name', 'Age', 'Major'], ["Susan", 19, "CS"]))
>>> d6 = dict([('Age', 19), ('Name', "Susan"), ('Major', "CS")])
```

Note: `zip` takes two equal-length collections and merges their corresponding elements into tuples.

ACCESSING THE DICTIONARY

To access a dictionary, simply index the dictionary by the key to obtain the value. An exception will be raised if the key is not in the dictionary.

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1['Age']
19
>>> d1['Name']
'Susan'
```

UPDATING A DICTIONARY

Simply assign a key:value pair to modify it or add a new pair. The del keyword can be used to delete a single key:value pair or the whole dictionary. The clear() method will clear the contents of the dictionary.

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1['Age'] = 21
>>> d1['Year'] = "Junior"
>>> d1
{'Age': 21, 'Name': 'Susan', 'Major': 'CS', 'Year': 'Junior'}
>>> del d1['Major']
>>> d1
{'Age': 21, 'Name': 'Susan', 'Year': 'Junior'}
>>> d1.clear()
>>> d1
{}
```

BUILT-IN DICTIONARY METHODS

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
>>> d1.has_key('Age') # True if key exists
True
>>> d1.has_key('Year') # False otherwise
False
>>> d1.keys() # Return a list of keys
['Age', 'Name', 'Major']
>>> d1.items() # Return a list of key:value pairs
[('Age', 19), ('Name', 'Susan'), ('Major', 'CS')]
>>> d1.values() # Returns a list of values
[19, 'Susan', 'CS']
```

Note: in, not in, pop(key), and popitem() are also supported.

ORDERED DICTIONARY

Dictionaries do not remember the order in which keys were inserted. An ordered dictionary implementation is available in the collections module. The methods of a regular dictionary are all supported by the OrderedDict class.

An additional method supported by OrderedDict is the following:

```
OrderedDict.popitem(last=True) # pops items in LIFO order
```

ORDERED DICTIONARY

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

EXCEPTIONS

Errors that are encountered during the execution of a Python program are *exceptions*.

```
>>> print spam
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

There are a number of built-in exceptions, which are listed [here](#).

HANDLING EXCEPTIONS

Explicitly handling exceptions allows us to control otherwise undefined behavior in our program, as well as alert users to errors. Use try/except blocks to catch and recover from exceptions.

```
>>> while True:
...     try:
...         x = int(raw_input("Enter a number: "))
...     except ValueError:
...         print("Oops !! That was not a valid number. Try again.")
...
Enter a number: two
Oops !! That was not a valid number. Try again.
Enter a number: 100
```

HANDLING EXCEPTIONS

- First, the try block is executed. If there are no errors, except is skipped.
- If there are errors, the rest of the try block is skipped.
 - Proceeds to except block with the matching exception type.
- Execution proceeds as normal.

```
>>> while True:
...     try:
...         x = int(raw_input("Enter a number: "))
...     except ValueError:
...         print("Oops !! That was not a valid number. Try again.")
...
Enter a number: two
Oops !! That was not a valid number. Try again.
Enter a number: 100
```

HANDLING EXCEPTIONS

- If there is no except block that matches the exception type, then the exception is unhandled and execution stops.

```
>>> while True:
...     try:
...         x = int(input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.")
...
Enter a number: 3/0
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
  File "<string>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Note our
change to
input here!

HANDLING EXCEPTIONS

The try/except clause options are as follows:

Clause form

```
except:
except name:
except name as value:
instance
except (name1, name2):
except (name1, name2) as value:
instance
else:
finally:
```

Interpretation

Catch all (or all other) exception types
Catch a specific exception only
Catch the listed exception and its
Catch any of the listed exceptions
Catch any of the listed exceptions and its
Run if no exception is raised
Always perform this block

HANDLING EXCEPTIONS

There are a number of ways to form a try/except block.

```
>>> while True:
...     try:
...         x = int(raw_input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.")
...     except (TypeError, IOError) as e:
...         print(e)
...     else:
...         print("No errors encountered!")
...     finally:
...         print("We may or may not have encountered errors...")
...
```

RAISING AN EXCEPTION

Use the raise statement to force an exception to occur. Useful for diverting a program or for raising custom exceptions.

```
try:
    raise IndexError("Index out of range")
except IndexError as ie:
    print("Index Error occurred: ", ie)
```

Output:
Index Error occurred: Index out of range

CREATING AN EXCEPTION

Make your own exception by creating a new exception class derived from the *Exception* class (we will be covering classes soon).

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e
...
My exception occurred, value: 4
```

ASSERTIONS

Use the `assert` statement to test a condition and raise an error if the condition is false.

```
>>> assert a == 2
```

is equivalent to

```
>>> if not a == 2:
...     raise AssertionError()
```

STRINGS

We've already introduced the string data type a few lectures ago. Strings are subtypes of the sequence data type.

Strings are written with either single or double quotes encasing a sequence of characters.

```
s1 = 'This is a string!'
s2 = "Python is so awesome."
```

Note that there is no character data type in Python. A character is simply represented as a string with one character.

ACCESSING STRINGS

As a subtype of the sequence data type, strings can be accessed element-wise as they are technically just sequences of character elements.

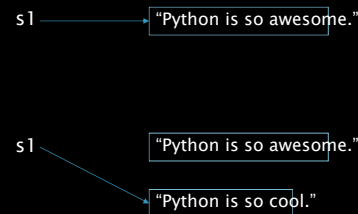
We can index with typical bracket notation, as well as perform slicing.

```
>>> s1 = "This is a string!"
>>> s2 = "Python is so awesome."
>>> print s1[3]
s
>>> print s2[5:15]
n is so aw
```

MODIFYING STRINGS

Strings are *immutable* – you cannot update the value of an existing string object. However, you can reassign your variable name to a new string object to perform an “update”.

```
>>> s1 = "Python is so awesome."
>>> s1 = "Python is so cool."
```



MODIFYING STRINGS

Alternatively, we could have done the following:

```
>>> s1 = "Python is so awesome."
>>> s1 = s1[:13] + "cool."
```

This will create a substring “Python is so”, which is concatenated with “cool.”, stored in memory and associated with the name `s1`.

The “+” operator can be used with two string objects to concatenate them together. The “*” operator can be used to concatenate multiple copies of a single string object.

We also have `in` and `not in` available for testing character membership within a string.

ESCAPE CHARACTERS

As a side note, there are a number of escape characters supported by Python strings. The most common ones are:

- ‘\n’ – newline
- ‘\s’ – space
- ‘\t’ – tab

BUILT-IN STRING METHODS

Python includes a number of built-in string methods that are incredibly useful for string manipulation. Note that these *return* the modified string value; we cannot change the string’s value in place because they’re immutable!

```

• s.upper() and s.lower()
>>> s1 = "Python is so awesome."
>>> print s1.upper()
PYTHON IS SO AWESOME.
>>> print s1.lower()
python is so awesome.
  
```

BUILT-IN STRING METHODS

• `s.isalpha()`, `s.isdigit()`, `s.isalnum()`, `s.isspace()` - return True if string *s* is composed of alphabetic characters, digits, either alphabetic and/or digits, and entirely whitespace characters, respectively.

• `s.islower()`, `s.isupper()` - return True if string *s* is all lowercase and all uppercase, respectively.

```
>>> "WHOA".isupper()
True
>>> "12345".isdigit()
True
>>> " \n ".isspace()
True
>>> "hello!".isalpha()
False
```

BUILT-IN STRING METHODS

• `str.split([sep[, maxsplit]])` - Split *str* into a list of substrings. The *sep* argument indicates the delimiting string (defaults to consecutive whitespace). The *maxsplit* argument indicates the maximum number of splits to be done (default is -1).

• `str.rsplit([sep[, maxsplit]])` - Split *str* into a list of substrings, starting from the right.

• `str.strip([chars])` - Return a copy of the string *str* with leading and trailing characters removed. The *chars* string specifies the set of characters to remove (default is whitespace).

• `str.rstrip([chars])` - Return a copy of the string *str* with only trailing characters removed.

BUILT-IN STRING METHODS

```
>>> "Python programming is fun!".split()
['Python', 'programming', 'is', 'fun!']
>>> "555-867-5309".split('-')
['555', '867', '5309']
>>> "***Python programming is fun***".strip('*')
'Python programming is fun'
```

BUILT-IN STRING METHODS

• `str.capitalize()` - returns a copy of the string with the first character capitalized and the rest lowercase.

• `str.center(width[, fillchar])` - centers the contents of the string *str* in field-size *width*, padded by *fillchar* (defaults to a blank space). See also `str.ljust()` and `str.rjust()`.

• `str.count(sub[, start[, end]])` - return the number of non-overlapping occurrences of substring *sub* in the range *[start, end]*. Can use slice notation here.

• `str.endswith(suffix[, start[, end]])` - return True if the string *str* ends with suffix, otherwise return False. Optionally, specify a substring to test. See also `str.startswith()`.

BUILT-IN STRING METHODS

```
>>> "i LoVe pYtHoN".capitalize()
'I love python'
>>> "centered".center(20, '*')
'*****centered*****'
>>> "mississippi".count("iss")
2
>>> "mississippi".count("iss", 4, -1)
1
>>> "mississippi".endswith("ssi")
False
>>> "mississippi".endswith("ssi", 0, 8)
True
```

BUILT-IN STRING METHODS

- `str.find(sub[, start[, end]])` - return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the slice *str[start:end]*. Return -1 if *sub* is not found. See also `str.rfind()`.
- `str.index(sub[, start[, end]])` - identical to `find()`, but raises a *ValueError* exception when substring *sub* is not found. See also `str.rindex()`.
- `str.join(iterable)` - return a string that is the result of concatenating all of the elements of *iterable*. The *str* object here is the delimiter between the concatenated elements.
- `str.replace(old, new[, count])` - return a copy of the string *str* where all instances of the substring *old* are replaced by the string *new* (up to *count* number of times).

BUILT-IN STRING METHODS

```
>>> "whenever".find("never")
3
>>> "whenever".find("what")
-1
>>> "whenever".index("what")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> "-".join(['555', '867', '5309'])
'555-867-5309'
>>> " ".join(['Python', 'is', 'awesome'])
'Python is awesome'
>>> "whenever".replace("ever", "ce")
'whence'
```

THE STRING MODULE

Additional built-in string methods may be found [here](#).

All of these built-in string methods are methods of any string object. They do not require importing any module or anything - they are part of the core of the language.

There is a string module, however, which provides some additional useful string tools. It defines useful string constants, the string formatting class, and some deprecated string functions which have mostly been converted to methods of string objects.

STRING CONSTANTS

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

STRING CONSTANTS

```
>>> import string
>>> string.lowercase #locale-dependent
'abcdefghijklmnopqrstuvwxyz'
>>> string.uppercase #locale-dependent
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.letters # lowercase+uppercase
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.octdigits
'01234567'
>>> print string.punctuation
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

ADVANCED FUNCTIONS AND OOP

FUNCTIONS

Before we start, let's talk about how name resolution is done in Python: When a function executes, a new namespace is created (locals). New namespaces can also be created by modules, classes, and methods as well.

LEGB Rule: How Python resolves names.

- Local namespace.
- Enclosing namespaces: check nonlocal names in the local scope of any enclosing functions from inner to outer.
- Global namespace: check names assigned at the top-level of a module file, or declared global in a def within the file.
- `__builtins__`: Names python assigned in the built-in module.
- If all fails: `NameError`.

FUNCTIONS AS FIRST-CLASS OBJECTS

We noted a few lectures ago that functions are *first-class objects* in Python. What exactly does this mean?

In short, it basically means that whatever you can do with a variable, you can do with a function. These include:

- Assigning a name to it.
- Passing it as an argument to a function.
- Returning it as the result of a function.
- Storing it in data structures.
- etc.

FUNCTION FACTORY

a.k.a. Closures.

As first-class objects, you can wrap functions within functions.

Outer functions have free variables that are bound to inner functions.

A closure is a function object that remembers values in enclosing scopes regardless of whether those scopes are still present in memory.

```
def make_inc(x):
    def inc(y):
        # x is closed in
        # the definition of inc
        return x + y
    return inc

inc5 = make_inc(5)
inc10 = make_inc(10)

print(inc5(5)) # returns 10
print(inc10(5)) # returns 15
```

CLOSURE

Closures are hard to define so follow these three rules for generating a closure:

1. We must have a nested function (function inside a function).
2. The nested function must refer to a value defined in the enclosing function.
3. The enclosing function must return the nested function.

DECORATORS

Wrappers to existing functions.

You can extend the functionality of existing functions without having to modify them.

```
def say_hello(name):
    return "Hello, " + str(name) + "!"

def p_decorate(func):
    def func_wrapper(name):
        return "<p>" + func(name) + "</p>"
    return func_wrapper

my_say_hello = p_decorate(say_hello)
print my_say_hello("John")
# Output is: <p>Hello, John!</p>
```

DECORATORS

Wrappers to existing functions.

You can extend the functionality of existing functions without having to modify them.

Closure

```
def say_hello(name):
    return "Hello, " + str(name) + "!"

def p_decorate(func):
    def func_wrapper(name):
        return "<p>" + func(name) + "</p>"
    return func_wrapper

my_say_hello = p_decorate(say_hello)
print my_say_hello("John")
# Output is: <p>Hello, John!</p>
```

DECORATORS

So what kinds of things can we use decorators for?

- Timing the execution of an arbitrary function.
- Memoization – cacheing results for specific arguments.
- Logging purposes.
- Debugging.
- Any pre- or post- function processing.

DECORATORS

Python allows us some nice syntactic sugar for creating decorators.

```
def say_hello(name):
    return "Hello, " + str(name) + "!"

def p_decorate(func):
    def func_wrapper(name):
        return "<p>" + func(name) + "</p>"
    return func_wrapper
```

Notice here how we have to explicitly decorate say_hello by passing it to our decorator function.

```
my_say_hello = p_decorate(say_hello)
print my_say_hello("John")
# Output is: <p>Hello, John!</p>
```

DECORATORS

Python allows us some nice syntactic sugar for creating decorators.

```
def p_decorate(func):
    def func_wrapper(name):
        return "<p>" + func(name) + "</p>"
    return func_wrapper
```

Some nice syntax that does the same thing, except this time I can use say_hello instead of assigning a new name.

```
@p_decorate
def say_hello(name):
    return "Hello, " + str(name) + "!"

print say_hello("John")
# Output is: <p>Hello, John!</p>
```

DECORATORS

You can also stack decorators with the closest decorator to the function definition being applied first.

```
@div_decorate
@p_decorate
@strong_decorate
def say_hello(name):
    return "Hello, " + str(name) + "!"

print say_hello("John")
# Outputs <div><p><strong>Hello, John!</strong></p></div>
```

DECORATORS

We can also pass arguments to decorators if we'd like.

```
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<" + tag_name + ">" + func(name) + "</" + tag_name + ">"
        return func_wrapper
    return tags_decorator

@tags("p")
def say_hello(name):
    return "Hello, " + str(name) + "!"

print say_hello("John") # Output is: <p>Hello, John!</p>
```

DECORATORS

We can also pass arguments to decorators if we'd like.

```
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<" + tag_name + ">" + func(name) + "</" + tag_name + ">"
        return func_wrapper
    return tags_decorator

@tags("p")
def say_hello(name):
    return "Hello, " + str(name) + "!"

print say_hello("John")
```

Closure!

DECORATORS

We can also pass arguments to decorators if we'd like.

```
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<" + tag_name + ">" + func(name) + "</" + tag_name + ">"
        return func_wrapper
    return tags_decorator

@tags("p")
def say_hello(name):
    return "Hello, " + str(name) + "!"

print say_hello("John")
```

More Closure!

ACCEPTS EXAMPLE

Let's say we wanted to create a general purpose decorator for the common operation of checking validity of function argument types.

```
import math
def complex_magnitude(z):
    return math.sqrt(z.real**2 + z.imag**2)

>>> complex_magnitude("hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "accepts_test.py", line 4, in complex_magnitude
    return math.sqrt(z.real**2 + z.imag**2)
AttributeError: 'str' object has no attribute 'real'
>>> complex_magnitude(1+2j)
2.23606797749979
```

ACCEPTS EXAMPLE

```
def accepts(*arg_types):
    def arg_check(func):
        def new_func(*args):
            for arg, arg_type in zip(args, arg_types):
                if type(arg) != arg_type:
                    print "Argument", arg, "is not of type", arg_type
                    break
            else:
                func(*args)
        return new_func
    return arg_check
```

Check out accepts_test.py!

OOP IN PYTHON

Python is a multi-paradigm language and, as such, supports OOP as well as a variety of other paradigms.

If you are familiar with OOP in C++, for example, it should be very easy for you to pick up the ideas behind Python's class structures.

CLASS DEFINITION

Classes are defined using the *class* keyword with a very familiar structure:

```
class ClassName(object):
    <statement-1>
    .
    .
    <statement-N>
```

There is no notion of a header file to include so we don't need to break up the creation of a class into declaration and definition. We just declare and use it!

CLASS OBJECTS

Let's say I have a simple class which does not much of anything at all.

```
class MyClass(object):
    """A simple example class docstring"""
    i = 12345
    def f(self):
        return 'hello world'
```

I can create a new instance of MyClass using the familiar function notation.

```
x = MyClass()
```

CLASS OBJECTS

I can access the attributes and methods of my object in the following way:

```
>>> x = MyClass()
>>> x.i
12345
>>> x.f()
'hello world'
```

We can define the special method `__init__()` which is automatically invoked for new instances (initializer).

```
class MyClass(object):
    """A simple example class"""
    i = 12345
    def __init__(self):
        print "I just created a MyClass object!"
    def f(self):
        return 'hello world'
```

CLASS OBJECTS

Now, when I instantiate a MyClass object, the following happens:

```
>>> y = MyClass()
I just created a MyClass object!
```

We can also pass arguments to our `__init__` function:

```
>>> class Complex(object):
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

DATA ATTRIBUTES

Like local variables in Python, there is no need for a data attribute to be declared before use.

```
>>> class Complex(object):
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
>>> x.r_squared = x.r**2
>>> x.r_squared
9.0
```

DATA ATTRIBUTES

We can add, modify, or delete attributes at will.

```
x.year = 2016 # Add an 'year' attribute.
x.year = 2017 # Modify 'year' attribute.
del x.year    # Delete 'year' attribute.
```

There are also some built-in functions we can use to accomplish the same tasks.

```
hasattr(x, 'year')    # Returns true if year attribute exists
getattr(x, 'year')    # Returns value of year attribute
setattr(x, 'year', 2017) # Set attribute year to 2015
delattr(x, 'year')    # Delete attribute year
```

VARIABLES WITHIN CLASSES

Generally speaking, variables in a class fall under one of two categories:

- **Class variables**, which are shared by all instances.
- **Instance variables**, which are unique to a specific instance.

```
>>> class Dog(object):
...     kind = 'canine' # class var
...     def __init__(self, name):
...         self.name = name # instance var
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind # shared by all dogs
'canine'
>>> e.kind # shared by all dogs
'canine'
>>> d.name # unique to d
'Fido'
>>> e.name # unique to e
'Buddy'
```

VARIABLES WITHIN CLASSES

Be careful when using mutable objects as class variables.

```
>>> class Dog(object):
>>>     tricks = [] # mutable class variable
>>>     def __init__(self, name):
>>>         self.name = name
>>>     def add_trick(self, trick):
>>>         self.tricks.append(trick)
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks # unexpectedly shared by all
['roll over', 'play dead']
```

VARIABLES WITHIN CLASSES

To fix this issue, make it an instance variable instead.

```
>>> class Dog(object):
>>>     def __init__(self, name):
>>>         self.name = name
>>>         self.tricks = []
>>>     def add_trick(self, trick):
>>>         self.tricks.append(trick)
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

BUILT-IN ATTRIBUTES

Besides the class and instance attributes, every class has access to the following:

- `__dict__`: dictionary containing the object's namespace.
- `__doc__`: class documentation string or None if undefined.
- `__name__`: class name.
- `__module__`: module name in which the class is defined. This attribute is `"__main__"` in interactive mode.
- `__bases__`: a possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

METHODS

We can call a method of a class object using the familiar function call notation.

```
>>> x = MyClass()
>>> x.f()
'hello world'
```

Perhaps you noticed, however, that the definition of `MyClass.f()` involves an argument called `self`.

Calling `x.f()` is equivalent to calling `MyClass.f(x)`.

```
class MyClass(object):
    """A simple example class"""
    i = 12345
    def __init__(self):
        print "I just created a MyClass object!"
    def f(self):
        return 'hello world'
```

FRACTION EXAMPLE

Check out Bob Myers' simple fraction class [here](#).

Let's check out an equivalent simple class in Python (`frac.py`).

FRACTION EXAMPLE

```
>>> import frac
>>> f1 = frac.Fraction()
>>> f2 = frac.Fraction(3,5)
>>> f1.get_numerator()
0
>>> f1.get_denominator()
1
>>> f2.get_numerator()
3
>>> f2.get_denominator()
5
```

FRACTION EXAMPLE

```
>>> f2.evaluate()
0.6
>>> f1.set_value(2,7)
>>> f1.evaluate()
0.2857142857142857
>>> f1.show()
2/7
>>> f2.show()
3/5
>>> f2.input()
2/3
>>> f2.show()
2/3
```

PET EXAMPLE

Here is a simple class that defines a Pet object.

```
class Pet(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def get_name(self):
        return self.name
    def get_age(self):
        return self.age
    def __str__(self):
        return "This pet's name is " + str(self.name) + str(self.age)
```

The `__str__` built-in function defines what happens when I print an instance of Pet. Here I'm overriding it to print the name.

PET EXAMPLE

Here is a simple class that defines a Pet object.

```
class Pet(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def get_name(self):
        return self.name
    def get_age(self):
        return self.age
    def __str__(self):
        return "This pet's name is " + str(self.name)
```

```
>>> from pet import Pet
>>> mypet = Pet('Ben', '2')
>>> print mypet
This pet's name is Ben
>>> mypet.get_name()
'Ben'
>>> mypet.get_age()
2
```

INHERITANCE

Now, let's say I want to create a Dog class which inherits from Pet. The basic format of a derived class is as follows:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    ...
    <statement-N>
```

In the case of BaseClass being defined elsewhere, you can use `module_name.BaseClassName`.

INHERITANCE

Here is an example definition of a Dog class which inherits from Pet.

```
class Dog(Pet):
    pass
```

The pass statement is only included here for syntax reasons. This class definition for Dog essentially makes Dog an alias for Pet.

INHERITANCE

We've inherited all the functionality of our Pet class, now let's make the Dog class more interesting.

```
>>> from dog import Dog
>>> mydog = Dog('Ben', 2)
>>> print mydog
This pet's name is Ben
>>> mydog.get_name()
'Ben'
>>> mydog.get_age()
2
```

```
class Dog(Pet):
    pass
```

INHERITANCE

For my Dog class, I want all of the functionality of the Pet class with one extra attribute: breed. I also want some extra methods for accessing this attribute.

```
class Dog(Pet):
    def __init__(self, name, age, breed):
        Pet.__init__(self, name, age)
        self.breed = breed
    def get_breed(self):
        return self.breed
```

INHERITANCE

For my Dog class, I want all of the functionality of the Pet class with one extra attribute: breed. I also want some extra methods for accessing this attribute.

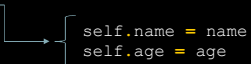
```
class Dog(Pet):
    def __init__(self, name, age, breed): ← Overriding initialization function
        Pet.__init__(self, name, age)
        self.breed = breed
    def get_breed(self):
        return self.breed
```

Python resolves attribute and method references by first searching the derived class and then searching the base class.

INHERITANCE

For my Dog class, I want all of the functionality of the Pet class with one extra attribute: breed. I also want some extra methods for accessing this attribute.

```
class Dog(Pet):
    def __init__(self, name, age, breed):
        Pet.__init__(self, name, age)
        self.breed = breed
    def get_breed(self):
        return self.breed
```



```
self.name = name
self.age = age
```

We can call base class methods directly using `BaseClassName.method(self, arguments)`. Note that we do this here to extend the functionality of Pet's initialization method.

INHERITANCE

```
>>> from dog import Dog
>>> mydog = Dog('Ben', 2, 'Maltese')
>>> print mydog
This pet's name is Ben
>>> mydog.get_age()
2
>>> mydog.get_breed()
'Maltese'
```

```
class Dog(Pet):
    def __init__(self, name, age, breed):
        Pet.__init__(self, name, age)
        self.breed = breed
    def get_breed(self):
        return self.breed
```

INHERITANCE

Python has two notable built-in functions:

- `isinstance(obj, cls)` returns true if *obj* is an instance of *cls* (or some class derived from *cls*).
- `issubclass(class, classinfo)` returns true if *class* is a subclass of *classinfo*.

```
>>> from pet import Pet
>>> from dog import Dog
>>> mydog = Dog('Ben', 2, 'Maltese')
>>> isinstance(mydog, Dog)
True
>>> isinstance(mydog, Pet)
True
>>> issubclass(Dog, Pet)
True
>>> issubclass(Pet, Dog)
False
```

MULTIPLE INHERITANCE

You can derive a class from multiple base classes like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    ...
    <statement-N>
```

Attribute resolution is performed by searching DerivedClassName, then Base1, then Base2, etc.

PRIVATE VARIABLES

There is no strict notion of a private attribute in Python.

However, if an attribute is prefixed with a single underscore (e.g. `_name`), then it should be treated as private. Basically, using it should be considered bad form as it is an implementation detail.

To avoid complications that arise from overriding attributes, Python does perform *name mangling*. Any attribute prefixed with two underscores (e.g. `__name`) and no more than one trailing underscore is automatically replaced with `__classname__name`.

Bottom line: if you want others developers to treat it as private, use the appropriate prefix.

NAME MANGLING

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.update(iterable)
    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

class MappingSubclass(Mapping):
    def update(self, keys, values):
        for item in zip(keys, values):
            self.items_list.append(item)
```

What's the problem here?

NAME MANGLING

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.update(iterable)
    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

class MappingSubclass(Mapping):
    def update(self, keys, values):
        for item in zip(keys, values):
            self.items_list.append(item)
```

What's the problem here?

The update method of Mapping accepts one iterable object as an argument.

The update method of MappingSubclass however, accepts keys and values as arguments.

Because MappingSubclass is derived from Mapping and we haven't overridden the `__init__` method, we will have an error when the `__init__` method calls `update` with a single argument.

NAME MANGLING

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.update(iterable)
    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

class MappingSubclass(Mapping):
    def update(self, keys, values):
        for item in zip(keys, values):
            self.items_list.append(item)
```

To be clearer, because MappingSubclass inherits from Mapping but does not provide a definition for `__init__`, we implicitly have the following `__init__` method.

```
def __init__(self, iterable):
    self.items_list = []
    self.update(iterable)
```

NAME MANGLING

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.update(iterable)
    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)
```

```
class MappingSubclass(Mapping):
    def update(self, keys, values):
        for item in zip(keys, values):
            self.items_list.append(item)
```

This `__init__` method references an `update` method. Python will simply look for the most local definition of `update` here.

```
def __init__(self, iterable):
    self.items_list = []
    self.update(iterable)
```

NAME MANGLING

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.update(iterable)
    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)
```

```
class MappingSubclass(Mapping):
    def update(self, keys, values):
        for item in zip(keys, values):
            self.items_list.append(item)
```

The signatures of the `update` call and the `update` definition do not match. The `__init__` method depends on a certain implementation of `update` being available. Namely, the `update` defined in `Mapping`.

```
def __init__(self, iterable):
    self.items_list = []
    self.update(iterable)
```

NAME MANGLING

```
>>> import map
>>> x = map.MappingSubclass([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "map.py", line 4, in __init__
    self.update(iterable)
TypeError: update() takes exactly 3 arguments (2 given)
```

NAME MANGLING

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)
    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)
    __update = update # private copy of original update() method

class MappingSubclass(Mapping):
    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

NAME MANGLING

```
>>> import map
>>> x = map.MappingSubclass([1,2,3])
>>> x.items_list
[1, 2, 3]
>>> x.update(['key1', 'key2'], ['val1', 'val2'])
>>> x.items_list
[1, 2, 3, ('key1', 'val1'), ('key2', 'val2')]
```

STRUCTS IN PYTHON

You can create a struct-like object by using an empty class.

```
>>> class Struct:
...     pass
...
>>> node = Struct()
>>> node.label = 4
>>> node.data = "My data string"
>>> node.next = Struct()
>>> next_node = node.next
>>> next_node.label = 5
>>> print node.next.label
5
```

EMULATING METHODS

You can create custom classes that emulate methods that have significant meaning when combined with other Python objects.

The statement `print >>` typically prints to the file-like object that follows. Specifically, the file-like object needs a `write()` method. This means I can make any class which, as long as it has a `write()` method, is a valid argument for this `print` statement.

```
>>> class Random:
...     def write(self, str_in):
...         print "The string to write is: " + str(str_in)
>>> someobj = Random()
>>> print >> someobj, "whatever"
The string to write is: whatever
```

CUSTOM EXCEPTIONS

We mentioned in previous lectures that exceptions can also be custom-made. This is done by creating a class which is derived from the Exception base class.

```
class MyException(Exception):
    def __init__(self, value):
        self.parameter = value
    def __str__(self):
        return self.parameter

>>> from myexcept import MyException
>>> try:
...     raise MyException("My custom error message.")
... except MyException as e:
...     print "Error: " + str(e)
...
Error: My custom error message.
```

ITERABLES, ITERATORS, AND GENERATORS

Before we move on to the standard library (in particular, the `itertools` module), let's make sure we understand iterables, iterators, and generators.

An *iterable* is any Python object with the following properties:

- It can be looped over (e.g. lists, strings, files, etc).
- Can be used as an argument to `iter()`, which returns an iterator.
- Must define `__iter__()` (or `__getitem__()`).

ITERABLES, ITERATORS, AND GENERATORS

Before we move on to the standard library (in particular, the `itertools` module), let's make sure we understand iterables, iterators, and generators.

An *iterator* is a Python object with the following properties:

- Must define `__iter__()` to return itself.
- Must define the `next()` method to return the next value every time it is invoked.
- Must track the "position" over the container of which it is an iterator.

ITERABLES, ITERATORS, AND GENERATORS

A common iterable is the list. Lists, however, are not iterators. They are simply Python objects for which iterators may be created.

```
>>> a = [1, 2, 3, 4]
>>> # a list is iterable - it has the __iter__ method
>>> a.__iter__
<method-wrapper '__iter__' of list object at 0x014E5D78>
>>> # a list doesn't have the next method, so it's not an iterator
>>> a.next
AttributeError: 'list' object has no attribute 'next'
>>> # a list is not its own iterator
>>> iter(a) is a
False
```

ITERABLES, ITERATORS, AND GENERATORS

The `listiterator` object is the iterator object associated with a list. The iterator version of a `listiterator` object is itself, since it is already an iterator.

```
>>> # iterator for a list is actually a 'listiterator' object
>>> ia = iter(a)
>>> ia
<listiterator object at 0x014DF2F0>
>>> # a listiterator object is its own iterator
>>> iter(ia) is ia
True
```

ITERATORS

How does this magic work?

```
for item in [1, 2, 3, 4]:
    print item
```

ITERATORS

How does this magic work?

The for statement calls the iter() function on the sequence object. The iter() call will return an iterator object (as long as the argument has a built-in __iter__ function) which defines next() for accessing the elements one at a time.

Let's do it manually:

```
>>> mylist = [1, 2, 3, 4]
>>> it = iter(mylist)
>>> it
<listiterator object at 0x2af6add16090>
>>> it.next()
1
>>> it.next()
2
>>> it.next()
3
>>> it.next()
4
>>> it.next() # Raises StopIteration Exception
```

ITERABLES, ITERATORS, AND GENERATORS

```
>>> mylist = [1, 2, 3, 4]
>>> for item in mylist:
...     print item

>>> mylist = [1, 2, 3, 4]
>>> i = iter(mylist) # i = mylist.__iter__()
>>> print i.next()
1
>>> print i.next()
2
>>> print i.next()
3
>>> print i.next()
4
>>> print i.next()
# StopIteration Exception Raised
```

↑ Is equivalent to →

ITERATORS

Let's create a custom iterable object.

```
class Even:
    def __init__(self, data):
        self.data = data
        self.index = 0
    def __iter__(self):
        return self
    def next(self):
        if self.index >= len(self.data):
            raise StopIteration
        ret = self.data[self.index]
        self.index = self.index + 2
        return ret
```

ITERATORS

Let's create a custom iterable object.

```
>>> from even import Even
>>> evenlist = Even(range(0,10))
>>> iter(evenlist)
<even.Even instance at 0x2ad24d84a128>
>>> for item in evenlist:
...     print item
...
0
2
4
6
8
```

ITERABLES, ITERATORS, AND GENERATORS

Generators are a way of defining iterators using a simple function notation.

Generators use the `yield` statement to return results when they are ready, but Python will remember the context of the generator when this happens.

Even though generators are not technically iterator objects, they can be used wherever iterators are used.

Generators are desirable because they are *lazy*: they do no work until the first value is requested, and they only do enough work to produce that value. As a result, they use fewer resources, and are usable on more kinds of iterables.

GENERATORS

An easy way to create "iterators". Use the `yield` statement whenever data is returned. The generator will pick up where it left off when `next()` is called.

```
def even(data):
    for i in range(0, len(data), 2):
        yield data[i]

>>> for elem in even(range(0,10)):
...     print elem
...
0
2
4
6
8
```

ITERABLES, ITERATORS, AND GENERATORS

```
def count_generator():
    n = 0
    while True:
        yield n
        n = n + 1
```

```
>>> counter = count_generator()
>>> counter
<generator object count_generator at 0x...>
>>> next(counter)
0
>>> next(counter)
1
>>> iter(counter)
<generator object count_generator at 0x...>
>>> iter(counter) is counter
True
>>> type(counter)
<type 'generator'>
```


ITERABLES, ITERATORS, AND GENERATORS

There are also generator comprehensions, which are very similar to list comprehensions.

```
>>> l1 = [x**2 for x in range(10)] # list
>>> g1 = (x**2 for x in range(10)) # gen
```

Equivalent to:

```
def gen(exp):
    for x in exp:
        yield x**2

g1 = gen(iter(range(10)))
```

LOGGING AND UNIT TEST

LOGGING

Logging is an essential practice for non-trivial applications in which events are recorded for potential diagnostic use in the future.

Logging can be used to record the following kinds of items:

- Errors: any exceptions that are raised can be logged when they are caught.
- Significant events: for example, when an administrator logs into a system.
- Data Handled: for example, a request came into the system with x, y, z parameters.

Logging can provide useful information about the state of the program during a crash or help a developer understand why the program is exhibiting some kind of behavior.

LOGGING

The Python Standard Library actually comes with a standard logging module. It is a particularly good decision to use this module because it can include messages generated from any other package that also uses this library.

As usual, you can use the logging module by using the import logging statement.

LOGGING

Here's a simple example of some logging calls.

```
>>> import logging
>>> logging.critical("Imminent fatal error!")
CRITICAL:root:Imminent fatal error!
>>> logging.error("Could not perform some function but still running.")
ERROR:root:Could not perform some function but still running.
>>> logging.warning("Disk space low...")
WARNING:root:Disk space low...
```

LOGGING

There are five logging levels which should be used accordingly to reflect the severity of the event being logged.

Level	Use
Debug	For development.
Info	Messages confirming expected behavior.
Warning	Warnings about future issues or benign unexpected behavior.
Error	Something unexpected happened and software is not working properly as a result.
Critical	Something unexpected happened and software is not running anymore as a result.

The logging.log(level, msg) methods are the easiest ways to quickly log a message with a given severity level. You can also use logging.log(level, msg).

LOGGING

Note that the info and debug messages are not logged. This is because the default logging level - the level at which messages must be logged - is warning and higher.

```
>>> import logging
>>> logging.critical("Imminent fatal error!")
CRITICAL:root:Imminent fatal error!
>>> logging.error("Could not perform some function but still running.")
ERROR:root:Could not perform some function but still running.
>>> logging.warning("Disk space low...")
WARNING:root:Disk space low...
>>> logging.info("Everything seems to be working ok.")
>>> logging.debug("Here's some info that might be useful to debug with.")
```

LOGGING

It is not typical to log directly in standard output. That might be terrifying to your client. You should at least direct logging statements to a file. Take the module logtest.py for example:

```
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.critical("Imminent fatal error!")
logging.error("Could not perform some function but still running.")
logging.warning("Disk space low...")
logging.info("Everything seems to be working ok.")
logging.debug("Here's some info that might be useful to debug with.")
```

LOGGING

It is not typical to log directly in standard output. That might be terrifying to your client. You should at least direct logging statements to a file. Take the module `logtest.py` for example:

```
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.critical("Imminent fatal error!")
logging.error("Could not perform some function but still running.")
logging.warning("Disk space low...")
logging.info("Everything seems to be working ok.")
logging.debug("Here's some info that might be useful to debug with.")
```

Log debug
messages and
higher

LOGGING

After running `logtest.py`, we have a logfile called `example.log` with the following contents:

```
CRITICAL:root:Imminent fatal error!
ERROR:root:Could not perform some function but still running.
WARNING:root:Disk space low...
INFO:root:Everything seems to be working ok.
DEBUG:root:Here's some info that might be useful to debug with.
```

LOGGING

```
import logging
import sys

for arg in sys.argv:
    if arg[1:] == "--loglevel=":
        loglvl = arg[1:].upper()
    else:
        loglvl = "INFO"

logging.basicConfig(filename='example.log', level=getattr(logging, loglvl))
logging.critical("Imminent fatal error!")
logging.error("Could not perform some function but still running.")
logging.warning("Disk space low...")
logging.info("Everything seems to be working ok.")
logging.debug("Here's some info that might be useful to debug with.")
```

An even better approach would be to allow the loglevel to be set as an argument to the program itself.

LOGGING

Now, I can specify what the logging level should be without changing the code – this is a more desirable scenario. If I do not set the logging level, it will default to INFO.

```
~$ python logtest.py --loglevel=warning
~$ more example.log
CRITICAL:root:Imminent fatal error!
ERROR:root:Could not perform some function but still running.
WARNING:root:Disk space low...
```

LOGGING

The `logging.basicConfig()` function is the simplest way to do basic global configuration for the logging system.

Any calls to `info()`, `debug()`, etc will call `basicConfig()` if it has not been called already. Any subsequent calls to `basicConfig()` have no effect.

Argument	Effect
filename	File to which logged messages are written.
filemode	Mode to open the filename with. Defaults to 'a' (append).
format	Format string for the messages.
datefmt	Format string for the timestamps.
level	Log level messages and higher.
stream	For StreamHandler objects.

LOGGING

If your application contains multiple modules, you can still share a single log file. Let's say we have a module `driver.py` which uses another module `mymath.py`.

```
import logging
import mymath

def main():
    logging.basicConfig(filename='example.log', level=logging.DEBUG)
    logging.info("Starting.")
    x = mymath.add(2, 3)
    logging.info("Finished with result " + str(x))

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

LOGGING

The `mymath.py` module also logs some message but note that we do not have to reconfigure the logging module. All the messages will log to the same file.

```
import logging

def add(num1, num2):
    logging.debug("num1 is " + str(num1) + " and num2 is " + str(num2))
    logging.info("Adding.")
    return num1 + num2
```

Note the logging of variable data here.

LOGGING

This behavior gives us some insight into the reason why additional calls `basicConfig()` have no effect. The first `basicConfig()` call made during the execution of the application is used to direct logging for all modules involved – even if they have their own `basicConfig()` calls.

```
~$ python driver.py
~$ more example.log
INFO:root:Starting.
DEBUG:root:num1 is 2 and num2 is 3
INFO:root:Adding.
INFO:root:Finished with result 5
```

LOGGING

You can modify the format of the message string as well. Typically, it's useful to include the level, timestamp, and message content.

```
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG,
                    format='%(asctime)s: %(levelname)s: %(message)s')
logging.info("Some important event just happened.")
```

```
~? python logtest.py
~? more example.log
2015-6-11 11:41:42,612:INFO:Some important event just happened.
```

LOGGING

All of the various formatting options can be found [here](#).

This is really just a very basic usage of the logging module, but its definitely enough to log a small project.

Advanced logging features give you a lot more control over when and how things are logged – most notably, you could implement a rotating series of log files rather than one very large logfile which might be difficult to search through.

AUTOMATED TESTING

Obviously, after you write some code, you need to make sure it works. There are pretty much three ways to do this, as pointed out by Ned Batchelder:

- Automatically test your code.
- Manually test your code.
- Just ship it and wait for clients to complain about your code.

The last is...just not a good idea. The second can be downright infeasible for a large project. That leaves us with automated testing.

TESTING

Let's say we have the following module with two simple functions.

```
even.py

def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False
```

TESTING

The simplest way to test is to simply pop open the interpreter and try it out.

```
even.py
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False

>>> import even
>>> even.even(2)
True
>>> even.even(3)
False
>>> even.pos_even(2)
True
>>> even.pos_even(3)
False
>>> even.pos_even(-2)
False
>>> even.pos_even(-3)
False
```

TESTING

This method is time-consuming and not repeatable. We'll have to redo these steps manually anytime we make changes to the code.

```
even.py
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False

>>> import even
>>> even.even(2)
True
>>> even.even(3)
False
>>> even.pos_even(2)
True
>>> even.pos_even(3)
False
>>> even.pos_even(-2)
False
>>> even.pos_even(-3)
False
```

TESTING

We can store the testing statements inside of a module and run them anytime we want to test. But this requires us to manually "check" the correctness of the results.

```
even.py
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False

test_even.py
import even
print "even.even(2) = ", even.even(2)
print "even.even(3) = ", even.even(3)
print "even.pos_even(2) = ", even.pos_even(2)
print "even.pos_even(3) = ", even.pos_even(3)
print "even.pos_even(-2) = ", even.pos_even(-2)
print "even.pos_even(-3) = ", even.pos_even(-3)
```

TESTING

We can store the testing statements inside of a module and run them anytime we want to test. But this requires us to manually "check" the correctness of the results.

```
even.py
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False

$ python test_even.py
even.even(2) = True
even.even(3) = False
even.pos_even(2) = True
even.pos_even(3) = False
even.pos_even(-2) = False
even.pos_even(-3) = False
```

TESTING

Let's use assert statements to our advantage. Now, when we test, we only need to see if there were any AssertionError exceptions raised. No output means all tests passed!

```
test_even.py
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False

import even
assert even.even(2) == True
assert even.even(3) == False
assert even.pos_even(2) == True
assert even.pos_even(3) == False
assert even.pos_even(-2) == False
assert even.pos_even(-3) == False
```

TESTING

Let's use assert statements to our advantage. Now, when we test, we only need to see if there were any AssertionError exceptions raised. No output means all tests passed!

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False
```

```
$ python test_even.py
$
```

However, one error will halt our testing program entirely so we can only pick up one error at a time. We could wrap assertions into try/except statements but now we're starting to do a lot of work for testing.

There must be a better way!

UNITTEST

The unittest module in the Standard Library is a framework for writing unit tests, which specifically test a *small* piece of code in isolation from the rest of the codebase.

Test-driven development is advantageous for the following reasons:

- Encourages modular design.
- Easier to cover every code path.
- The actual process of testing is less time-consuming.

UNITTEST

Here's an example of the simplest usage of unittest.

```
test_even.py
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        assert even.even(2) == True

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

```
even.py
def even(num):
    if abs(num)%2 == 0:
        return True
    return False
```

```
$ python test_even.py
```

```
.
-----
Ran 1 test in 0.000s

OK
```

UNITTEST

Here's an example of the simplest usage of unittest.

test_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        assert even.even(2) == True

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

even.py

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False
```

\$ python test_even.py

```
.
```

Ran 1 test in 0.000s

OK

All tests are defined in methods (which must start with "test_") of some custom class that derives from unittest.TestCase.

UNITTEST BEHIND THE SCENES

By calling unittest.main() when we run the module, we are giving control to the unittest module. It will create a new instance of EvenTest for every test method we have so that they can be performed in isolation.

test_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        assert even.even(2) == True

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

What unittest does:

```
testcase = EvenTest()

try:
    testcase.test_is_two_even()
except AssertionError:
    [record failure]
else:
    [record success]
```

UNITTEST

test_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        self.assertTrue(even.even(2))
    def test_two_positive(self):
        self.assertTrue(even.pos_even(2))

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

even.py

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num > 0:
            return False
        return True
    return False
```

We've added some new tests along with some new source code. A couple things to notice: our source code has a logical error in it and we're no longer manually asserting. We're using unittest's nice assert methods.

UNITTEST

\$ python test_even.py

.F

```
=====
FAIL: test_two_positive (__main__.EvenTest)
```

Traceback (most recent call last):

```
File "test_even.py", line 8, in test_two_positive
    self.assertTrue(even.pos_even(2))
```

AssertionError: False is not true

Ran 2 tests in 0.000s

FAILED (failures=1)

Extra information given to us by unittest's special assertion method.

UNITTEST

The unittest module defines a ton of assertion methods:

- `assertEqual(f, s)`
- `assertNotEqual(f, s)`
- `assertIn(f, s)`
- `assertIs(f, s)`
- `assertGreater(f, s)`
- `assertRaises(exc, f, ...)`
- etc.

UNITTEST

test_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        self.assertTrue(even.even(2))
    def test_two_positive(self):
        self.assertTrue(even.pos_even(2))

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

even.py

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False
```

UNITTEST

```
$ python test_even.py
```

```
..
```

```
-----
Ran 2 tests in 0.000s
```

```
OK
```

```
$
```

UNITTEST

We incrementally add unit test functions and run them – when they pass, we add more code and develop the unit tests to assert correctness. Do not remove unit tests as you pass them.

Also, practice unit testing as much as you can. Do not wait until it is absolutely necessary.

As with logging, there is a lot more to unit testing that we're not covering here so definitely look up the docs and read articles about unit testing in Python to learn more about the advanced features.

Ned Batchelder also has a relevant unit testing talk from PyCon '14. Check it out [here](#).

DOCUMENTATION

Being able to properly document code, especially large projects with multiple contributors, is incredibly important.

Code that is poorly-documented is sooner thrown-out than agonized over. So make sure your time is well-spent and document your code for whoever may need to see it in the future!

Python, as to be expected, has a selection of unique Python-based documenting tools and as a Python developer, it is important that you be familiar with at least one of them.