Basic Language

Reserved Keywords

Although Python will not stop you, do not use these as variable or function names.

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

or these either

```
abs all any apply basestring bin bool buffer bytearray
bytes callable chr classmethod cmp coerce compile complex
delattr dict dir divmod enumerate eval execfile file
filter float format frozenset getattr globals hasattr
hash hex id input int intern isinstance issubclass iter
len list locals long map max min next object oct open ord
pow print property range raw_input reduce reload repr
reversed round set setattr slice sorted staticmethod str
sum super tuple type unichr unicode vars xrange zip
```

Numbers

Python has common-sense number-handling capabilities.

```
>>> 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> 7/3 # Integer division returns the floor:
2
>>> 7/-3 # ext integer down
-3
>>> width = 20
>>> height = 5*9
>>> width * height
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2 # float numerator
3.5
>>> from __future__ import division # no more default integer division!
>>> 7/2
3.5
>>> 7 // 2 # double slash gives integer division
3
```

You can also cast among the numeric types in a common-sense way:

```
>>> int(1.33333)
1
```

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

Complex Numbers

Python has rudimentary support for complex numbers.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
>>> a=1.5+0.5j
>>> a.real # the dot notation gets an attribute
1.5
>>> a.imag
0.5
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

Strings

String-handling is very well developed and highly optimized in Python. We just cover the main points here. First, single or double quotes define a string and there is no precedence relationship between single and double quotes.

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Python strings have C-style escape characters for newlinews, tabs, etc.

```
>>> print """Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
     """

>>> # the 'r' makes this a 'raw' string
>>> hello = r"This is a rather long string containing\n\ several lines of text much as you w
>>> print hello
This is a rather long string containing\n\ several lines of text much as you would do in C.
>>> # otherwise, you get the newline \n
>>> hello = "This is a rather long string containing\n\ several lines of text much as you wo
>>> print hello
This is a rather long string containing
 several lines of text much as you would do in C.
>>> u'this a unicode string  µ ±'  # the 'u' makes it a unicode string
>>> u'this a unicode string \xb5 \xb1' # using hex-codes
```

Slicing Strings

Python is a zero-indexed language (like C). The color `:` character denotes slicing.

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # Everything except the last two characters
'Hel'
>>> word[0] = 'Q' # strings are IMMUTABLE
Traceback (most recent call last):
   ...
TypeError: 'str' object does not support item assignment
```

String Operations

Some basic numerical operations work with strings.

```
>>> 'hey '+'you' # concatenate with plus operator
'hey you'
>>> 'hey '*3 # multiply duplicates strings
'hey hey hey '
```

Python has a built-in and very powerful regular expression module for string manipulations.

Substitutions

String substitution creates new strings.

```
>>> x = 'This is a string'
>>> print x.replace('string','newstring')
This is a newstring
>>> print x # x hasn't changed
This is a string
```

Formatting Strings

There are *so* many ways to format strings in Python, but here is the simplest that follows C-language `sprintf` conventions in conjunction with the modulo operator `%`.

```
>>> print 'this is a decimal number %d' % ( 10 )
this is a decimal number 10
>>> print 'this is a float %3.2f' % (10.33)
this is a float 10.33
>>> x = 1.03
>>> print 'this is a variable %e' % (x) # exponential format
this is a variable 1.030000e+00
```

Exercise

Exercise

In a separate file, write a program to print “Hello World!” 10 times.

1. Create a new file with the .py extension using your favorite editor
2. Save your file
3. Open a CMD terminal window
4. Run your program using

```
c:> python Your_file_name_here.py
```

and observe the output.

Could you write the same program using 1-line? Note you do not need any loops or anything beyond what we have just covered.

Basic Data Structures

Python provides many powerful data structures. The two most powerful and fundamental are the list and dictionary. We have already seen that strings are immutable. Python also provides immutable data structures.

Lists - generalized containers (mutable)

The `list` is an order-preserving general container.

```
>>> mix=[3,'tree',5.678,[8,4,2]];  mix                 # List creation
[3, 'tree', 5.6779999999999999, [8, 4, 2]]
>>> mix[1]                                             # Indexing individual elements
'tree'
>>> mix[0]
3
>>> mix[-2]                                            # Indexing from the right
5.6779999999999999
>>> mix[3]
[8, 4, 2]
>>> mix[3][1]                                          # Last element is itself a list
4
>>> mix[0]=666;  mix                                   # Mutable
[666, 'tree', 5.6779999999999999, [8, 4, 2]]
>>> submix=mix[0:3];  submix                           # Creating sublist
[666, 'tree', 5.6779999999999999]
>>> switch=mix[3]+submix; switch                       # + Operator
[8, 4, 2, 666, 'tree', 5.6779999999999999]
>>> len(switch)                                        # Built-in Function
6
>>> resize=[6.45,'SOFIA',3,8.2E6,15,14]; len(resize)
6
```

```
>>> resize[1:4]=[55]; resize; len(resize)              # Shrink a sublist
[6.4500000000000002, 55, 15, 14]
4
>>> resize[3]=['all','for','one']; resize; len(resize)
[6.4500000000000002, 55, 15, ['all', 'for', 'one']]
4
>>> resize[4]=2.87                                    # Cannot append this way
Traceback (most recent call last):
   ...
IndexError: list assignment index out of range
>>> temp=resize[:3]
>>> resize=temp+resize[3]; resize; len(resize)        # add to list
[6.4500000000000002, 55, 15, 'all', 'for', 'one']
6
>>> del resize[3]; resize; len(resize)                # delete an element
[6.4500000000000002, 55, 15, 'for', 'one']
5
>>> del resize[1:3]; resize; len(resize)              # delete a sublist
[6.4500000000000002, 'for', 'one']
3
```

Note

Sorting lists

```
>>> sorted([1,9,8,2]) # this sorts lists
[1, 2, 8, 9]
```

Exercise

Exercise

1. Given the following list

```
data=[5,4,6,1,9,0,3,9,2,7,10,8,4,7,1,2,7,6,5,2,8,2,0,1,1,1,2,10,6,2]
```

compute the average of this data. Hint: you need the `sum` built-in function.

Tuples – containers (immutable)

Tuples are another general purpose sequential container in Python, very similar to lists, but these are **immutable**. Tuples are delimited by commas (parentheses are grouping symbols).

```
>>> tuple([1,3,4])
(1, 3, 4)
>>> 1,3,4
(1, 3, 4)
```

```
>>> pets=('dog','cat','bird')  #Parentheses () create tuples
>>> pets[0]
'dog'
>>> pets + pets # addition
('dog', 'cat', 'bird', 'dog', 'cat', 'bird')
>>> pets*3
('dog', 'cat', 'bird', 'dog', 'cat', 'bird', 'dog', 'cat', 'bird')
>>> pets[0]='rat' # assignment not work!
TypeError: 'tuple' object does not support item assignment
```

Key Concept: Mutable vs. Immutable

Key Concept: Mutable vs. Immutable

In Python, the easiest way to think of variables is in terms of immutable (not changeable) or mutable (changeable).

```
>>> x = [1,3,4,['one',1.33]]       # lists are a mutable (i.e. changeable)
>>> x[0]='banana'                  # no problem changing this
>>> y = (1,3,4,['one',1.33])       # tuples are a immutable (i.e. not changeable)
>>> y[0]='banana'
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

```
>>> z=('banana',)+y[1:]           # I need to make a new tuple to change it
>>> print id(y)
25690000
>>> print id(z)
26030320
>>> print id(x)
26055664
>>> x[1]='here I go again'        # make a change (okay!)
>>> print id(x)
26055664
```

Dictionaries - The MOST IMPORTANT Python data structure

```
>>> top={'math':'Gauss','phys':'Newton','art':'Vemeer','phil':'Emerson',
... 'play':'Shakespeare','actor':'Kidman','direct':'Kubrick',
... 'author':'Hemmingway','bio':'Watson','group':'R.E.M'}
...
>>> len(top) #{ } are dictionary creation operators
10
>>> top['pres']='F.D.R.'
>>> top['bio']='Darwin'
>>> #can delete a key,value pair
>>> del top['actor']
>>> # another way of creating a dict
>>> x=dict(key='value',another_key=1.333,more_keys=[1,3,4,'one'])
>>> print x
{'another_key': 1.333, 'more_keys': [1, 3, 4, 'one'], 'key': 'value'}
>>> x={(1,3):'value'}     # any immutable type can be a valid key
>>> print x
{(1, 3): 'value'}
>>> x[(1,3)]='immutables can be keys'
```

[Advanced Tip] Unions of Dictionaries

What if you want to create a union of dictionaries in one-line?

```
>>> d1 = {'a':1, 'b':2, 'c':3}
>>> d2 = {'A':1, 'B':2, 'C':3}
>>> dict(d1,**d2) # combo of d1 and d2
{'A': 1, 'B': 2, 'C': 3, 'a': 1, 'b': 2, 'c': 3}
```

Pretty slick.

Sets

Python provides mathematical sets and corresponding operations with the `set()` data structure.

```
>>> print set([1,2,11,1]) # union-izes elements
set([1, 2, 11])
>>> print set([1,2,3]) & set([2,3,4]) # intersection
set([2, 3])
>>> print set([1,2,3]) and set([2,3,4]) # not work
set([2, 3, 4])
>>> print set([1,2,3]) ^ set([2,3,4]) # exclusive OR
set([1, 4])
>>> print set([1,2,3]) | set([2,3,4]) # OR
```

```
set([1, 2, 3, 4])
>>> print set([ [1,2,3],[2,3,4] ]) # no sets of lists (w/o more work)
Traceback (most recent call last):
TypeError: unhashable type: 'list'
```

More Python Data Structures

- Fast Non-Standard Data Structures for Python
- Python Set & Frozenset
- SortedContainers
- BloomFilter in python
- Static memory-efficient Trie-like structures for Python (2.x and 3.x) based on marisa-trie C++ library.
- An implementation of using the x-fast trie for fast predecessor search.
- Sieve Of Eratosthenes In Python
- GitHub - keon/algorithms: Minimal examples of data structures and algorithms in Python
- GitHub - donnemartin/interactive-coding-challenges: Huge update! Interactive Python coding interview challenges (algorithms and data structures). Includes Anki flashcards.

Basic Programming

Loops and Conditionals

There are two primary looping constructs in Python: the `for` loop and the `while` loop. The syntax of the `for` loop is straightforward:

```
>>> for i in range(3):
...     print i
```

Note the colon character at the end. This is your hint that the next line should be indented. The `for` loop iterates over items that provided by the `iterator`, which is the `range(3)` list in the above example. Python abstracts the idea of `iterable` out of the looping construction so that some Python objects are iterable all by themselves and are just waiting for an iteration provider like a `for` or `while` loop to get them going.

The `while` loop has a similar straightforward construct:

```
>>> i = 0
>>> while i < 10:
...     i += 1
...
```

Again, note that the presence of the colon character hints at indenting the following line. The `while` loop will continue until the boolean expression (i.e., `i<10`) evaluates `False`. Let's consider booleans and membership in Python.

- Loop through a list in Python and modify it - Stack Overflow
- List Chaining and Permutations Vlad Bezden

Logic and Membership

Python is a *truthy* language:

Only these are false:

- `None`

- `False`
- zero of any numeric type, for example, `0, 0L, 0.0, 0j`.
- any empty sequence, for example, `'', (), []`.
- any empty mapping, for example, `{}`.
- instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns the integer zero or bool value False.

```
>>> bool(1)
True
>>> bool([]) # empty list
False
>>> bool({}) # empty dictionary
False
>>> bool(0)
False
>>> bool([0,])
True
```

Python is syntactically clean about numeric intervals!

```
>>> 3.2 < 10 < 20
True
```

You can use disjunctions (`or`), negations (`not`), and conjunctions (`and`) also.

```
>>> 1 < 2 and 2 < 3 or 3 < 1
True
>>> 1 < 2 and not 2 > 3  or 1<3
True
```

It's advisable to use grouping parentheses for readability. You can use logic across iterables as in the following:

```
>>>  (1,2,3) < (4,5,6) # at least one True
True
>>>  (1,2,3) < (4,5,1) # logical disjunction across elements
True
```

I've never seen this idiom in the wild, though. Don't use relative comparison for Python strings (i.e., `'a'< 'b'`). It gets weird like that. Use string matching operations instead (i.e., `==`).

Membership testing uses the `in` keyword.

```
>>> 'on' in [22,['one','too','throw']]
False
>>> 'one' in [22,['one','too','throw']] # no recursion
False
>>> 'one' in [22,'one',['too','throw']]
True
>>> ['too','throw'] not in [22,'one',['too','throw']]
False
```

If you are testing membership across millions of elements it is **much** faster to use `set()` instead of list. For example,

```
>>> 'one' in {'one','two','three'}
>>> 'one' in set(['one','two','three'])
```

The `is` keyword is stronger than equality, because it checks if two objects are the *same*.

```
>>> x = 'this string'
>>> y = 'this string'
>>> print x is y
False
>>> print x==y
True
```

However, `is` is really checking the `id` of each of the items:

```
>>> x=y='this string'
>>> print id(x),id(y)
4281928416 4281928416
>>> print x is y
True
```

By virtue of this, the following idioms are common: `x is True`, `x is None`.

- python – Multiple repeated in keyword – Stack Overflow

Conditionals

Now that we understand boolean expressions, we can build conditional statements using `if`.

```
>>> if 1 < 2:
...     print 'one less than two'
```

There is an `else` and `elif`, but no `switch` statement.

```
>>> a = 10
>>> if a < 10:
...     print 'a less than 10'
... elif a < 20:
...    print 'a less than 20'
... else:
...     print 'a otherwise'
```

There is a one-liner for conditionals

```
>>> x = 1 if (1>2) else 3 # 1-line conditional
>>> print x
3
```

The `for` loop can have an `else` clause as in the following

```
>>> rangelist = range(10)
>>> print rangelist
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for number in rangelist:
...       # Check if number is one of
...       # the numbers in the tuple.
...       if number in (3, 4, 7, 9):
...           # "Break" terminates a for without
...           # executing the "else" clause.
...           break
...       else:
...           # "Continue" starts the next iteration
...           # of the loop. It's rather useless here,
...           # as it's the last statement of the loop.
...           continue
... else:
...       # The "else" clause is optional and is
...       # executed only if the loop didn't "break".
...       pass # Do nothing
```

This is good for recognizing when a loop exits normally or via `break`.

List Comprehensions

Collecting items over a loop is common that it is its own idiom in Python. That is,

```
>>> out=[] # initialize container
>>> for i in range(10): out.append(i**2)
>>> out
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This can be abbreviated as a list comprehension.

```
>>> [i**2 for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Conditional elements can be embedded in the list comprehension.

```
>>> [i**2 for i in range(10) if i % 2] # embedded conditional
[1, 9, 25, 49, 81]
```

These comprehensions also works as dictionaries and sets.

```
>>> {i:i**2 for i in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
>>> {i**2 for i in range(5)}
{0, 1, 4, 9, 16}
```

Exercise

Exercises

1. Build a dictionary to with keys as integers `0,1,2,3,4,5` and the corresponding squares of these integers as values.

2. Build a reverse-lookup for the dictionary in part `1`. That is, given the squared integer (i.e., the value of the above dictionary), find the corresponding square root from the dictionary (i.e., the key of the above dictionary).

Functions

There are two common ways to define functions. You can use the `def` keyword as in the following:

```
>>> def foo(): return 'I said foo'
>>> foo()
'I said foo'
```

Note that you need a `return` statement and that you need the parenthesis to actually invoke the function. Without the `return` statement, the functions returns the `None` singleton.

Functions are first-class objects.

```
>>> foo # just another Python object
<function __main__.foo>
```

Practically speaking, this means they can be manipulated like any other Python object — they can be put in containers and passed around without any special handling. Naturally, we want to supply arguments to our functions. There are two kinds of function arguments *positional* and *keyword*.

```
>>> def foo(x): # positional unnamed argument
...     return x*2
...
>>> foo(10)
20
```

Keyword arguments allow you to specify defaults.

```
>>> def foo(x=20): # keyword named argument
...     return 2*x
...
>>> foo(1)
2
>>> foo()
40
>>> foo(x=30)
60
>>>
>>> def foo(x=20,y=30):
...     return x+y
...
>>> foo(20,)
50
>>> foo(1,1)
2
>>> foo(y=12)
32
>>> help(foo)
Help on function foo in module __main__:
...
foo(x=20, y=30) # provides automatic documentation
```

Python makes it easy to include documentation for your functions using *docstrings*. The makes the `help` function more useful for functions.

```
>>> def foo(position=20,velocity=30): #using a documentation string docstring
...     '''position in m
...     velocity in m/s
...     '''
...     return x+y
>>> help(foo)
Help on function foo in module __main__:
...
foo(position=20, velocity=30)
    position in m
    velocity in m/s
```

Thus, by using meaningful argument and function names and including basic documentation in the *docstrings*, you can greatly improve the usability of your Python functions. Also, it is recommended to make function names verb-like (e.g., `get_this`, `compute_field`)

In addition to using the `def` statement, you can also create functions using `lambda`. These are sometimes called *anonymous* functions.

```
>>> f = lambda x:  x**2 # anonymous functions
>>> f(10)
100
>>> [lambda x: x, lambda x:x**2] # list of functions
[<function <lambda> at ...>, <function <lambda> at ...>]
>>> for i in  _:
...   print i(10)
10
100
```

So far, we have not made good use of `tuple`, but these data structures become very powerful when used with functions. This is because they allow you to separate the function arguments (i.e., function signature) from the function itself. This means you can pass them around and build function arguments and then later execute them with one or more functions.

Using tuples with functions

```
The usefulness of tuples derives from using them with functions

>>> args = (1,3,4)
>>> def foo(x,y,z): #defining
...   return x+y+z
>>> print foo(*args) # note the asterisk notation
8

This also works with keyword arguments

>>> def foo(x=1,y=2,z=3):
...   return x+y+z
>>> kwargs = {'x':10,'y':20,'z':30}
>>> print foo(**kwargs) # note the double asterisks
60
>>> kwargs = {'x':10,'y':20,'z':30,'h':100} # extra argument
>>> print foo(**kwargs) # note the double asterisks
Traceback (most recent call last):
...
TypeError: foo() got an unexpected keyword argument 'h'

This also works in combination

>>> def foo(x,y=2,z=3): # note the the first argument is positional
...    return x+y+z
>>> kwargs = {'y':20,'z':30}
>>> args = (1,)
>>> foo(*args,**kwargs)
51
```

Function Variable Scoping

Variables within functions or subfunctions are local to that respective scope. Global variables require special handling if they are going to be changed inside the function body.

```
>>> x=10 # outside function
>>> def foo():
...    return x
>>> def foo():
...    x=1 # defined inside function
...    return x
>>> def foo():
...   global x # define as global
...   x=20     # assign inside function scope
...   return x
>>> print x
10
```

Function keyword filtering

Using `**kwds` at the end allows functions to disregard keywords they don't use while filtering out (using the function signature) the keyword inputs that it *does* use.

```
def foo(x=1,y=2,z=3,**kwds):
   return x+y+z

def goo(x=10,**kwds):
   return foo(x=2*x,**kwds)

def moo(y=1,z=1,**kwds):
   return goo(x=z+y,z=z+1,q=10,**kwds)
```

This means I can call any of these with an unsupported keyword as in

```
moo(y=91,z=11,zeta_variable = 10,**kwds)
```

and the `zeta_variable` will be passed through unused because nobody in the calling sequence uses it. Thus, you can inject some other function in the calling sequence that *does* use this variable, without having to change the call signatures of any of the other functions. Using keyword arguments this way is very common when using Python to wrap other codes.

Here's another example where we can trace how each of the function signatures are satisfied and the rest of the keyword arguments are passed through.

```
def foo(x=1,y=2,**kwds):
    print 'foo: x = %d, y = %d, kwds=%r'%(x,y,kwds)
    print '\t',
    goo(x=x,**kwds)

def goo(x=10,**kwds):
    print 'goo: x = %d, kwds=%r'%(x,kwds)
    print '\t\t',
    moo(x=x,**kwds)

def moo(z=20,**kwds):
    print 'moo: z=%d, kwds=%r'%(z,kwds)
```

Then, calling the *foo* function as shown below:

```
foo(x=1,y=2,z=3,k=20)
```

Gives the following output:

```
foo: x = 1, y = 2, kwds={'k': 20, 'z': 3}
        goo: x = 1, kwds={'k': 20, 'z': 3}
                moo: z=3, kwds={'x': 1, 'k': 20}
```

Notice how the function signatures of each of the functions are satisfied and the rest of the keyword arguments are passed through.

Functional Programming Idioms

Although not a *real* functional programming language (i.e., Haskell), Python has useful functional idioms. These become important in parallel computing frameworks like PySpark, but have become deprecated in Python 3.x.

```
>>> print map( lambda x:  x**2 , range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [i**2 for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> reduce(lambda x,y:x+2*y,[0,1,2,3],0)
12
>>> filter(lambda x: x%2, range(10) )
[1, 3, 5, 7, 9]
>>> [i for i in range(10) if i %2 ]
[1, 3, 5, 7, 9]
```

Pay attention to the recursive problem that `reduce` solves because `reduce` is super-fast in Python. I've seen very tricky problems solved using `reduce` creatively. For example, the least common multiple algorithm can be effectively implemented using `reduce`, as shown:

```
def gcd(a, b):
    """Return greatest common divisor using Euclid's Algorithm."""
    while b:
        a, b = b, a % b
    return a

def lcm(a, b):
    """Return lowest common multiple."""
    return a * b // gcd(a, b)

def lcmm(*args):
    """Return lcm of args."""
    return reduce(lcm, args)
```

Beware default containers in functions

```
>>> def foo(x=[]): # using empty list as default
...     x.append(10)
...     return x

>>> foo() # maybe you expected this...
[10]
>>> foo() # ... but did you expect this...
[10, 10]
>>> foo() # ... or this? What's going on here?
[10, 10, 10]
```

Exercises

Exercises

1. Given the following list from last time,

   ```
   data=[5,4,6,1,9,0,3,9,2,7,10,8,4,7,1,2,7,6,5,2,8,2,0,1,1,1,2,10,6,2]
   ```

   remove all the duplicated elements from this list. Your output should be `[5,4,6,1,9,0,3,2,7,10,8]`. Note that the order is preserved.

2. Write a function that computes the sum of all positive integers up to a specified value, *n*.

3. Write a function that determines whether or not a strictly positive integer is *perfect*. A positive integer is perfect if the sum of all its proper divisors is equal to itself. For example, 28 is perfect because its proper divisors are `[1,2,4,7,14]` and the `sum([1,2,4,7,14]) = 28`.

4. Generate all subsets of a set. Note there will be `2**len(x)` subsets of the set `x`. For example, the set `[0,1,2]` has eight subsets: `{[],[0],[1],[2],[0,1],[1,2],[0,2],[0,1,2]}`. Note that you don't have to mind the ordering of each subset. Hint: use recursion. Hint: You probably want to use `list` instead of `set`.

5. Generate all ordered sub-lists for a given list. For example, `x = [0,1,2]` has sub-lists

   `[[0],[1],[2],[1,2],[0,1],[0,2],[0,1,2]]`

   Hint: use `slice`.

6. Implement a reverse dictionary lookup. For example, given the following dictionary:

   ```
   d = {i:(i%2) for i in range(10)}
   {0: 0, 1: 1, 2: 0, 3: 1, 4: 0, 5: 1, 6: 0, 7: 1, 8: 0, 9: 1}
   ```

   return a list of all the dictionary keys that map to 0 and 1, respectively. In this case,

   `{0: [0, 2, 4, 6, 8], 1: [1, 3, 5, 7, 9]}`

   Note that the output is a *dictionary* that has a list for each of the values of `d`.

7. Write a function to return `True` or `False` based on whether two intervals represented as tuples overlap. For example, `(0,1)` overlaps with `(0,10)` so the function should return `True`. Likewise, `(1,2)` does not overlap `(3,4)` so the function should return `False` .

File Input/Output

It's straightforward to read and write files using Python. The same pattern applies when writing to other objects, like sockets.

Basic file usage

The following is the traditional way to get I/O in Python. The modern way is to use (The with statement).

```
>>> f=open('myfile.txt','wb') # write-binary mode
>>> f.write('this is line 1')
>>> f.close()

>>> f=open('myfile.txt','ab') # append-binary mode
>>> f.write('this is line 2')
>>> f.close()

>>> f=open('myfile.txt','ab') # append-binary mode
>>> f.write('\nthis is line 3\n')  # put in newlines
>>> f.close()
>>> f=open('myfile.txt','ab') # append-binary mode
>>> f.writelines([ 'this is line 4\n', 'this is line 5\n'])  # put in newlines

>>> f=open('myfile.txt','rb') # read-binary mode
>>> print f.readlines()
['this is line 1this is line 2\n', 'this is line 3\n', 'this is line 4\n', 'this is line 5\n
```

Note

Python has many tools for handling file I/O at different levels of granularity. The `struct` module is good for pure binary reading and writing by interpreting strings as packed binary data. The mmap module is useful for bypassing the filesystem and using virtual memory for fast file access. The StringIO module allows read and write of strings as files.

Exercise

Exercise

1. Recall the data from last time

```
data=[5,4,6,1,9,0,3,9,2,7,10,8,4,7,1,2,7,6,5,2,8,2,0,1,1,1,2,10,6,2]
```

Write three columns to a file

```
data_value, data_value**2, (data_value+data_value**2)/3.
```

2. Download this corpus of 10,000 common English words and do the following:
   - Compute the average length of the words.
   - What is the longest word?
   - What is the longest word that starts with *s*
   - What is the most common starting letter?
   - What is the most common ending letter?
3. Using the same file as above, create a new file that consists of each consecutive non-overlapping sequence of five lines merged into one line. Here're the first 10 lines:

```
the of and to a
in for is on that
by this with i you
it not or be are
from at as your all
have new more an was
we will home can us
about if page my has
search free but our one
other do no information time
```

The better and modern way to deal with file I/O is to use the with statement (The with statement).

Serialization: Saving Complex objects

Serialization means packing Python objects to be shipped between separate Python processes or separate computers, say, through a network socket, for example. The multiplatform nature of Python means that one cannot be assured that the low-level attributes of Python objects (say, between platform types or Python versions) will remain consistent. This video provides an in-depth discussion of each step of the Python serialization process.

For the vast majority of situations, the following will work.

```
>>> import cPickle
>>> mylist = ["This", "is", 4, 13327]
>>> f=open ('myfile.dat','wb') # write-binary mode
>>> cPickle.dump(mylist, f)
>>> f.close()
>>> f=open ('myfile.dat','rb') # write-binary mode
>>> print cPickle.load(f)
['This', 'is', 4, 13327]
```

Note

You can also serialize strings and transport them with intermediate file creation, using sockets, or some other protocol.

Pickling a Function

The internal state of a function and how it's hooked into the Python process in which it was created makes it tricky to pickle functions. As with everything in Python, there are ways around this, depending on your use-case. The following links provide some context:

- Is there an easy way to pickle a python function (or otherwise serialize its code)?
- How to pickle a python function with its dependencies? - Stack Overflow
- Pyro - Python Remote Objects - documentation
- dill
- MessagePack: It's like JSON. but fast and small.
- Using Pythons Pathlib Module - Practical Business Python

Quoting the first link above, the idea is to use the `marshal` module to dump the function object in a binary format, write it to a file, and then reconstruct it on the other end using `types.FunctionType`. The downside of this technique is that it may not be compatible across different major Python versions, even if they are all CPython implementations.

```
>>> import marshal
>>> def foo(x): return x*x
>>> code_string = marshal.dumps(foo.func_code)
```

Then in the remote process (after transferring code_string):

```
>>> import marshal, types
>>> code = marshal.loads(code_string)
>>> func = types.FunctionType(code, globals(), "some_func_name")
>>> func(10)  # gives 100
```

See the links listed above for much more discussion on this topic. A way to do this that is designed to work with parallel computing is to use dill which is probably the easiest way to handle this. However, `dill` does not always work with Pypy and it hijacks all pickling after the import statement. For more fine-grained control of serialization using `dill`, do `dill.extend(False)` after importing `dill`.

```
>>> import dill
>>> import cPickle
>>> def foo(x): return x*x
>>> print dill.dumps(foo)
>>> print cPickle.dumps(foo)
```

Dealing with Errors

This is where Python really shines. More modern languages should adopt the Python approach to error handling. The Python approach is to ask for forgiveness rather than permission. This is the basic template.

```
>>> try:
...   # try something
... except:
...   # fix something
```

The above `except` block with capture and process any kind of exception that is thrown in the `try` block. Python provides a long list of built-in exceptions that you can catch and the ability to create your own exceptions, if needed. In addition to catching exceptions, you can raise your own exception using the `raise` statement. There is also an `assert` statement that can throw exceptions if certain statements are not `True` upon assertion (more on this later).

The following are some examples of how to use the exception handling powers of Python.

```
>>> def some_function():
...     try:
...         # Division by zero raises an exception
...         10 / 0
...     except ZeroDivisionError:
...         print "Oops, invalid."
...     else:
...         # Exception didn't occur, we're good.
...         pass
...     finally:
...         # This is executed after the code block is run
...         # and all exceptions have been handled, even
...         # if a new exception is raised while handling.
...         print "We're done with that."

>>> some_function()
Oops, invalid.
We're done with that.

>>> out = range (3)
>>> try:
...     10 / 0
... except ZeroDivisionError:
...     print 'I caught an attempt to divide by zero'
I caught an attempt to divide by zero
```

```
>>> try:
...     out[999] # raise IndexError
... except ZeroDivisionError:
...     print 'I caught an attempt to divide by zero'
Traceback (most recent call last):
...
IndexError: list index out of range

>>> try:
...     1/0 # raise ZeroDivisionError
...     out[999]
... except ZeroDivisionError:
...     print 'I caught an attempt to divide by zero but I did not try out[999]'
I caught an attempt to divide by zero but I did not try out[999]

>>> try:
...     1/0 # raise ZeroDivisionError
...     out[999]
... except IndexError:
...     print 'I caught an attempt to index something out of range'
Traceback (most recent call last):
...
ZeroDivisionError: integer division or modulo by zero

>>> try:  #nested exceptions
...   try: # inner scope
...      1/0
...   except IndexError:
...      print 'caught index error inside'
... except ZeroDivisionError as e:
...     print 'I caught an attempt to divide by zero inside somewhere'
I caught an attempt to divide by zero inside somewhere

>>> try:  #nested exceptions with finally clause
...   try:
...      1/0
...   except IndexError:
...      print 'caught index error inside'
...   finally:
...      print "I am working in inner scope"
... except ZeroDivisionError as e:
...     print 'I caught an attempt to divide by zero inside somewhere'
I am working in inner scope
I caught an attempt to divide by zero inside somewhere

>>> try:
...     1/0
... except (IndexError,ZeroDivisionError) as e:
...     if isinstance(e,ZeroDivisionError):
...        print 'I caught an attempt to divide by zero inside somewhere'
...     elif isinstance(e,IndexError):
...        print 'I caught an attempt to index something out of range'
I caught an attempt to divide by zero inside somewhere

>>> try: # more detailed arbitrary exception catching
...     1/0
... except Exception as e:
...     print type(e)
<type 'exceptions.ZeroDivisionError'>
```

Exercise

Exercise

1. Write a function that takes a single string character (i.e., 'a','b','c') as input and returns *True* or *False* if that character represents a valid integer.
2. Write a function to compute the area of the rectangle given the lengths of its sides, but you will only compute for rectangles, not squares, so raise an exception if otherwise.

3. Given a list of color hex-codes `['#FFAABB']`, write a function to convert these into a list of RGB-tuples. For example, `[(255,170,187)]` corresponds to the example above. Here is the list of hex-codes to convert:

```
['#FAEBD7', '#00FFFF', '#7FFFD4', '#F0FFFF', '#F5F5DC', '#FFE4C4',
'#000000', '#FFEBCD', '#0000FF', '#8A2BE2', '#A52A2A', '#DEB887',
'#5F9EA0', '#7FFF00', '#D2691E', '#FF7F50', '#6495ED', '#FFF8DC',
'#DC143C', '#00FFFF', '#00008B', '#133B63', '#104E8B', '#008B8B',
'#B8860B', '#A9A9A9', ]
```

Exercise

Given the following string

```
>>> data = """Mary had a little lamb
... its fleece was white as snow
... and everywhere that Mary went
... the lamb was sure to go"""
```

1. Compute the average length of the words in it.
2. Excluding repeated words, re-compute the average length of the words.

Exercise

What is inadvisable about the following loop?

```
>>> data = range(10)
>>> for i in data:
...   if i % 2: # odd
...       data.remove(i)
>>> print data
[0, 2, 4, 6, 8]
```

Power Python Features to Master

The under-appreciated `zip` function

Python has a built-in `zip` function that can combine iterables pair-wise.

```
>>> zip(range(3),'abc')
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> zip(range(3),'abc',range(1,4))
[(0, 'a', 1), (1, 'b', 2), (2, 'c', 3)]
```

The slick part is reversing this operation using the `*` operation,

```
>>> x = zip(range(3),'abc')
>>> print x
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> i,j = zip(*x)
>>> print i
(0, 1, 2)
>>> print j
('a', 'b', 'c')
```

When combined with `dict`, `zip` provide a powerful way to build Python dictionaries,

```
>>> k = range(10)
>>> v = range(10,20)
>>> dict(zip(k,v))
{0: 10, 1: 11, 2: 12, 3: 13, 4: 14, 5: 15, 6: 16, 7: 17, 8: 18, 9: 19}
```

The `max` function

The `max` function takes the maximum of a sequence.

```
>>> max([1,3,4])
4
```

If the items in the sequence are tuples, then the first item in the tuple is used for the ranking

```
>>> max([(1,2),(3,4)])
(3,4)
```

This function takes a `key` argument that controls how the items in the sequence are evaluated. For example, we can rank based on the second element in the tuple,

```
>>> max([(1,4),(3,2)],key=lambda i:i[1])
(1,4)
```

The `with` statement

Set up a context for subsequent code

```
class ControlledExecution:
    def __enter__(self):
        #set things up
        return thing
    def __exit__(self, type, value, traceback):
        #tear things down

with ControlledExecution() as thing:
     some code
```

Using with files:

```
>>> f = open("sample1.txt")
>>> f
<open file 'sample1.txt', mode 'r' at 0x00AE82F0>
>>> f.__enter__()
<open file 'sample1.txt', mode 'r' at 0x00AE82F0>
>>> f.read(1)
0
>>> f.__exit__(None, None, None)
>>> f.read(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

This is the new way to open close files that makes it harder to forget to close the files when you're done.

```
with open("x.txt") as f:
    data = f.read()
    #do something with data
```

Good discussion of with statement

- Understanding Python's WITH statement

`contextlib` – Module for Fast Context Construction

```
import contextlib
```

```
@contextlib.contextmanager
def my_context():
   print 'setting up '
   try:
      yield {} # can yield object if necessary for 'as' part
   except:
      print 'catch some errors here'
   finally:
      print 'tearing down'
```

```
>>> with my_context():
...    print 'I am in the context'
      setting up
      I am in the context
      tearing down

>>> with my_context():
...     raise  RuntimeError ('I am an error')
       setting up
       catch some errors here
       tearing down
```

Exercise

Exercise

Given a set of weights `{1,3,9,27}`, show that you can weigh any number between 1 and 40. In other words, using the set above and the addition and subtraction operations, construct any integer between 1 and 40 without re-using elements. For example, `4 = 1+1+1+1` is *not* acceptable.

For example,

```
8 = 9 - 1
```

```
10 = 1 + 9
```

Hint: see the itertools module.

Exercise

1. Implement the Sieve of Eratosthenes. In other words, create function that can compute all prime numbers up to a given fixed integer.
2. Implement the Bubble sort as a Python function.
3. Implement a sliding window for an arbitrary input list. The function should take the window width and the window increment as inputs and should produce a sequence of overlapping lists from the input list. For example, given *`x=range(15)`*, the following is the output given a window width of *`5`* and window increment of *`2`*.

   ```
   [[0, 1, 2, 3, 4],
    [2, 3, 4, 5, 6],
    [4, 5, 6, 7, 8],
    [6, 7, 8, 9, 10],
    [8, 9, 10, 11, 12],
    [10, 11, 12, 13, 14]]
   ```

4. Given an integer, print the next smallest and next largest number that have the same number of 1 bits in their binary representation. Hint:

   ```
   print '{0:03b}'.format(4)
   ```

   will print out the binary representation of an integer for the requested number of bits (i.e., above is for 3-bits).

5. Using the *random.randrange* function, generate 100 random integers between *0* and *9*. Count how many of these integers fit in the intervals *[0,2]*, *[3,7]*, *[8,9]*.

Advanced Language Features

Advanced String Formatting

Strings can be templated using dictionary keys,

```
>>> "this is the key's value: %(key)s " % {'key':'Value_from_dictionary'}
"this is the key's value: Value_from_dictionary "
```

It's okay if there are other unused entries in the dictionary:

```
>>> "this is the key's value: %(key)s " % {'key':'Value_from_dictionary','Another_entry':'An
"this is the key's value: Value_from_dictionary "
```

Using Advanced String Formatting allows for arbitrary collection of positional or keyword arguments using placeholders

```
>>> print "{0} {1} {2}".format('First_item',100,10.333)
First_item 100 10.333
>>> print "{1} {0} {2}".format('First_item',100,10.333) # change order
100 First_item 10.333
>>> print "{{0}} {1} {2}".format('First_item',100,10.333) # using double curly braces
{0} 100 10.333
```

You can also access attributes this way:

```
>>> print "{0.real} + {0.imag}".format(12+1j)
12.0 + 1.0
>>> print "{0.real:3.2f} + {0.imag:3.3e}".format(12+1j) # using format specifiers
12.00 + 1.000e+00
>>> print "{0[key]}".format({'key':'value'})
value
```

Dictionaries can also be accessed via key/value pair

```
>>> '%(key)s %(x)d'%dict(key='KEY',x=10)
'KEY 10'
```

More advanced string formatting material

- PyCon 2010: Advanced String Formatting
- Common string operations Python v2.7.2 documentation

Generators

Just-in-time memory efficient containers

- Produces a stream of on-demand values
- Only executes on `next()`
- `yield()` produces a value, but saves the function's state for later
- Consumable (use once-only)

```
>>> def generate_ints(N):
...   for i in range(N):
```

```
...     yield i # the yield makes the function a generator
>>> x=generate_ints (3)
>>> x.next()
0
>>> x.next()
1
>>> x.next()
2
>>> x.next()
Traceback (most recent call last):
...
StopIteration
```

What's happening here?

```
>>> generate_ints(3).next()
0
>>> generate_ints(3).next()
0
>>> generate_ints(3).next()
0
```

You can also iterate on these directly:

```
>>> for i in generate_ints(5): # no assignment necessary here
...     print i
0
1
2
3
4
```

Generators maintain an internal state

```
>>> def foo():
...   print 'hello'
...   yield 1
...   print 'world'
...   yield 2
>>> x = foo()
>>> x.next()
hello
1
>>> # do some other stuff here
>>> x.next() # pick up where I left off
world
2

>>> def pi_series(): # infinite series converges to pi
...    sm = 0
...    i = 1.0; j = 1
...    while True: # loops forever!
...       sm = sm + j/i
...       yield 4*sm
...       i = i + 2; j = j * -1
>>> x = pi_series()
>>> x.next()
4.0
>>> x.next()
2.666666666666667
>>> x.next()
3.4666666666666668
>>> x.next()
2.8952380952380956
>>> x.next()
3.3396825396825403
>>> x.next()
2.9760461760461765
```

```
>>> x.next()
3.2837384837384844
>>> gen=( i for i in range(3) )    # List comprehension style
```

You can also send to an existing generator:

```
>>> def foo():
...   while True:
...      line=(yield)
...      print line
>>> x= foo()
>>> x.next() # get it going
>>> x.send('I sent this to you')
I sent this to you
```

These can be daisy-chained also:

```
>>> def goo(target):
...   while True:
...      line=(yield)
...      target.send(line.upper()+'---')
>>> x= foo()
>>> y= goo(x)
>>> x.next() # get it going
>>> y.next() # get it going
>>> y.send('from goo to you')
FROM GOO TO YOU---
```

Generators can also be created as list comprehensions by changing the bracket notation to parentheses.

```
>>> x= (i for i in range(10))
>>> print type(x)
<type 'generator'>
```

Now, *x* is a generator. The *itertools* module is key to using generators effectively. For example, we can clone generators

```
>>> x= (i for i in range(10))
>>> import itertools as it
>>> y,=it.tee(x,1) # copy generator
>>> y.next() # step this one
>>> print zip(x,y)
[(1, 2), (3, 4), (5, 6), (7, 8)]
```

This delayed execution method becomes particularly useful when working with large data sets.

```
>>> x= (i for i in range(10))
>>> y = it.imap(lambda i:i**2,x)
>>> y
<itertools.imap at 0x48e0be0>
```

Note that *y* is also a generator and that nothing has been computed yet. You can also map functions onto sequences using *it.starmap* that would yield yet another a generator. Likewise, you can chain generators together by `it.chain`,

```
>>> x= (i for i in range(10))
>>> y= (i for i in range(10,20))
>>> z= (i for i in range(20,30))
>>> it.chain(x,y,z)
<itertools.chain at 0x7f8c581a6910>
```

Exercise

Exercise: Fibonacci numbers using generators

The Fibonacci numbers are defined by the following recursion: $F_n = F_{n-2} + F_{n-1}$ with initial values $F_1 = F_2 = 1$. Using generators, compute the first ten Fibonacci numbers, `[1,1,2,3,5,8,13,21,34,55,89]`.

Note

References

- A Curious Course on Coroutines and Concurrency
- Gevent Python network library that uses greenlet and libevent for easy and scalable concurrency
- Generators: Powering Iteration in Python

Decorators

Decorators are functions that make functions out of functions. It sounds redundant, but turns out to be very useful for uniting disparate concepts.

```
>>> def my_decorator(fn): # note that function as input
...   def new_function(*args,**kwargs): # we've seen these arguments before
...       print 'this runs before function'
...       return fn(*args,**kwargs) # return a function
...   return new_function
>>> def foo(x):
...   return 2*x
>>> goo = my_decorator(foo)

>>> foo (3)
6
>>> goo (3)
this runs before function
6
```

Decorators are useful for debugging

```
>>> def log_arguments(fn): # note that function as input
...   def new_function(*args,**kwargs): # we've seen these arguments before
...       print 'positional arguments:'
...       print args
...       print 'keyword arguments:'
...       print kwargs
...       return fn(*args,**kwargs) # return a function
...   return new_function

>>> @log_arguments # these are stackable also
... def foo(x,y=20):
...   return x*y

>>> foo (1,y=3)
positional arguments:
(1,)
keyword arguments:
{'y': 3}
3
```

Decorators are very useful for caches

```
def simple_cache(fn):
  cache = {}
  def new_fn(n):
     if n in cache:
        print 'FOUND IN CACHE; RETURNING'
        return cache[n]
     # otherwise, call function
```

```
    # & record value
    val = fn(n)
    cache[n] = val
    return val
  return new_fn

>>> def foo(x):
...   return 2*x
>>> goo = simple_cache(foo)
>>> print [goo(i) for i in range(5)]
[0, 2, 4, 6, 8]
>>> print [goo(i) for i in range(8)]
FOUND IN CACHE; RETURNING
FOUND IN CACHE; RETURNING
FOUND IN CACHE; RETURNING
FOUND IN CACHE; RETURNING
FOUND IN CACHE; RETURNING
[0, 2, 4, 6, 8, 10, 12, 14]
```

Some Python modules are distributed as decorators (e.g., the click module for creating commandline interfaces) to make it easy to bolt-on new functionality without changing the source code.

Decorators are also useful for executing certain functions in threads.

```
def run_async(func):
   from threading import Thread
   from functools import wraps
   @wraps(func)
   def async_func(*args, **kwargs):
      func_hl = Thread(target = func, args = args, kwargs = kwargs)
      func_hl.start()
      return func_hl
   return async_func
```

The `wrap` function from the `functools` module fixes the function signature. This decorator is useful when you have a small side-job (like a notification) that you want to run out of the main thread of execution. Let's write a simple function that does some fake *work*.

```
def sleepy(n=1,id=1):
    print 'item %d sleeping for %d seconds...'%(id,n)
    sleep(n)
    print 'item %d done sleeping for %d seconds'%(id,n)
```

Consider the following code block:

```
sleepy(1,1)
print 'I am here!'
sleepy(2,2)
print 'I am now here!'
```

With the corresponding sequence of printed outputs:

```
item 1 sleeping for 1 seconds...
item 1 done sleeping for 1 seconds
I am here!
item 2 sleeping for 2 seconds...
item 2 done sleeping for 2 seconds
I am now here!
```

Using the decorator, we can make asynchronous versions of this function:

```
@run_async
def sleepy(n=1,id=1):
   print 'item %d sleeping for %d seconds...'%(id,n)
```

```
    sleep(n)
    print 'item %d done sleeping for %d seconds'%(id,n)
```

And with the same block of statements above, we obtain the following sequence of printed outputs:

```
I am here!
item 1 sleeping for 1 seconds...
item 2 sleeping for 2 seconds...
I am now here!
item 1 done sleeping for 1 seconds
item 2 done sleeping for 2 seconds
```

Notice that the last print statement in the block actually executed **before** the individual functions were completed. That is because the main thread of execution is handling those print statements while the separate threads are sleeping for different amounts of time. In other words, in the first example the last statement is *blocked* by the previous statements and has to wait for them to finish before it can do the final print. In the second case, there is no blocking so it can get to the last statement right away while the other work goes on in separate threads.

Decorators in Python can form *closures*.

```
def return_func(a):
    'scalar multiply decorator'
    def foo(x):
        return a*x # a is not in function scope but is trapped
    return foo
```

Note that the variable that is passed into the inner function is *not* in the scope of the inner function. To see this work,

```
>>> f2 = return_func(2)
>>> f3 = return_func(3)
>>> f2(10)
20
>>> f3(10)
30
```

Coroutines

Decorators and generators can be combined to create a coroutine. Practically speaking, you can think of a coroutine as a special kind of state-preserving function that must be externally driven.

```
def coroutine(fn):
    'decorator to auto-start generators'
    def kickstart(*args,**kwargs):
        x = fn(*args,**kwargs)
        x.next()
        return x
    return kickstart
```

Now, we can use this to create an auto-starting generators.

```
@coroutine
def capword():
    out = ''
    while True:
        word = yield out
        out=word.upper()
```

Then, we can use this as

```
>>> c = capword()
>>> c.send('hello')
'HELLO'
>>> c.send('bye')
'BYE'
```

We can define another generator,

```
@coroutine
def reverse_words():
   out = ''
   while True:
      word = yield out
      out=word[::-1]
```

These can be composed as in the following:

```
>>> r = reverse_words()
>>> r.send(c.send('hello'))
'OLLEH'
```

We can chain these by passing in the coroutine.

```
@coroutine
def reverse_words(other):
   out = ''
   while True:
      word = yield other.send(out)
      out=word[::-1]
```

Then,

```
>>> c = capword()
>>> r = reverse_words(c)
>>> r.send('foo')
'OOF'
```

This coroutine method is the heart of the `async` framework in Python 3.

Note

- PyCon 2010:Decorators From Basics to Class Decorators to Decorator Libraries
- Joblib: running Python functions as pipeline jobs
- Understanding the underscore(__) of Python

Iteration and Iterables

Iterators permit finer control for looping constructs.

```
>>> a = range(3)
>>> hasattr(a,'__iter__')
True
>>> # generally speaking, returns object that supports iteration
>>> iter(a)
>>> hasattr(_,'__iter__')
False
>>> for i in a: #use iterables in loops
...   print i
0
1
2
```

You can also use `iter()` with functions to create sentinels

```
>>> x=1
>>> def sentinel():
...     global x
...     x+=1
...     return x
>>> for k in iter(sentinel,10): # stops when x = 10
...     print k
2
3
4
5
6
7
8
9
>>> print x
10
```

You can use this with a file object `f` as in `iter(f.readline,'')` which will read lines from the file until the end.

- Python Iterators: A Step-By-Step Introduction
- Sorting Algorithms Visualized in Python
- OmkarPathak/pygorithm: A Python module for learning all major algorithms
- Understanding the underscore(_) of Python
- A Python Internals Adventure
- Intersection of Non-Empty Sets in Python | hack.write()
- satwikkansal/wtfpython: A collection of interesting, subtle, and tricky Python snippets.
- Why in python 0, 0 == (0, 0) equals (0, False) - Stack Overflow
- Things you need to know about garbage collection in Python | Artem Golubin
- norvig/pytudes: Python programs to practice or demonstrate skills.
- Queues in Python
- python - Why does chained assignment work this way? - Stack Overflow