Python Practice Book

Release 2014-08-10

Anand Chitipothu

February 25, 2017

Contents

Welcome to **Python Practice Book**.

CHAPTER 1

About this Book

This book is prepared from the training notes of Anand Chitipothu.

Anand conducts Python training classes on a semi-regular basis in Bangalore, India. Checkout out the upcoming trainings if you are interested.

Table of Contents

Getting Started

Running Python Interpreter

Python comes with an interactive interpreter. When you type `python` in your shell or command prompt, the python interpreter becomes active with a >>> prompt and waits for your commands.

```
$ python
Python 2.7.1 (r271:86832, Mar 17 2011, 07:02:35)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now you can type any valid python expression at the prompt. python reads the typed expression, evaluates it and prints the result.

```
>>> 42
42
>>> 4 + 2
6
```

Problem 1: Open a new Python interpreter and use it to find the value of `2 + 3`. >>> 5

Running Python Scripts

Open your text editor, type the following text and save it as `hello.py`.

```
print "hello, world!"
```

And run this program by calling `python hello.py`. Make sure you change to the directory where you saved the file before doing it.

```
anand@bodhi ~$ python hello.py
hello, world!
anand@bodhi ~$
```

C:\Users\Amiz\Documents\Sandbox\Python\practice>python hello.py
hello, World!

Text after # character in any line is considered as comment.

```
# This is helloworld program
# run this as:
#    python hello.py
print "hello, world!"
```

C:\Users\Amiz\Documents\Sandbox\Python\practice>python hello.py
hello, World!
hello, World!

Problem 2: Create a python script to print `hello, world!` four times.

Problem 3: Create a python script with the following text and see the output.

```
1 + 2
```

If it doesn't print anything, what changes can you make to the program to print the value?

Assignments

One of the building blocks of programming is associating a name to a value. This is called assignment. The associated name is usually called a *variable*.

```
>>> x = 4
>>> x * x
16
```

In this example `x` is a variable and it's value is `4`.

If you try to use a name that is not associated with any value, python gives an error message.

```
>>> foo
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'foo' is not defined
>>> foo = 4
>>> foo
4
```

If you re-assign a different value to an existing variable, the new value overwrites the old value.

```
>>> x = 4
>>> x
4
>>> x = 'hello'
>>> x
'hello'
```

It is possible to do multiple assignments at once.

```
>>> a, b = 1, 2
>>> a
1
>>> b
2
>>> a + b
3
```

Swapping values of 2 variables in python is very simple.

```
>>> a, b = 1, 2
>>> a, b = b, a
>>> a
2
>>> b
1
```

When executing assignments, python evaluates the right hand side first and then assigns those values to the variables specified in the left hand side.

Problem 4: What will be output of the following program.

```
x = 4
y = x + 1
x = 2
print x, y
```

Problem 5: What will be the output of the following program.

```
x, y = 2, 6
x, y = y, x + 2
print x, y
```

Problem 6: What will be the output of the following program.

```
a, b = 2, 3
c, b = a, c + 1
print a, b, c
```

Numbers

We already know how to work with numbers.

```
>>> 42
42
>>> 4 + 2
6
```

Python also supports decimal numbers.

```
>>> 4.2
4.2
>>> 4.2 + 2.3
6.5
```

Python supports the following operators on numbers.

- + addition
- − subtraction
- * multiplication
- / division
- ** exponent
- % remainder

Let's try them on integers.

```
>>> 7 + 2
9
>>> 7 - 2
5
>>> 7 * 2
14
>>> 7 / 2
3
>>> 7 ** 2
49
>>> 7 % 2
1
```

result shown in pthon 3.8 is 3.5

If you notice, the result `7 / 2` is `3` not `3.5`. It is because the `/` operator when working on integers, produces only an integer. Lets see what happens when we try it with decimal numbers:

```
>>> 7.0 / 2.0
3.5
>>> 7.0 / 2
3.5
>>> 7 / 2.0
3.5
```

The operators can be combined.

```
>>> 7 + 2 + 5 - 3
11
>>> 2 * 3 + 4
10
```

It is important to understand how these compound expressions are evaluated. The operators have precedence, a kind of priority that determines which operator is applied first. Among the numerical operators, the precedence of operators is as follows, from low precedence to high.

- `+`, `-`
- `*`, `/`, `%`
- `**`

When we compute `2 + 3 * 4`, `3 * 4` is computed first as the precedence of `*` is higher than `+` and then the result is added to 2.

```
>>> 2 + 3 * 4
14
```

We can use parenthesis to specify the explicit groups.

```
>>> (2 + 3) * 4
20
```

All the operators except `**` are left-associcate, that means that the application of the operators starts from left to right.

```
1 + 2 + 3 * 4 + 5
  ↓
  3   + 3 * 4 + 5
          ↓
  3   +   12  + 5
      ↓
      15      + 5
              ↓
             20
```

Strings

Strings what you use to represent text.

Strings are a sequence of characters, enclosed in single quotes or double quotes.

```
>>> x = "hello"
>>> y = 'world'
>>> print x, y
hello world
```

There is difference between single quotes and double quotes, they can used interchangebly.

Multi-line strings can be written using three single quotes or three double quotes.

```
x = """This is a multi-line string
written in
three lines."""
print x

y = '''multi-line strings can be written
using three single quote characters as well.
The string can contain 'single quotes' or "double quotes"
in side it.'''
print y
```

Functions

Just like a value can be associated with a name, a piece of logic can also be associated with a name by defining a function.

```
>>> def square(x):
...     return x * x
...
>>> square(5)
25
```

The body of the function is indented. Indentation is the Python's way of grouping statements.

The `...` is the secondary prompt, which the Python interpreter uses to denote that it is expecting some more input.

The functions can be used in any expressions.

```
>>> square(2) + square(3)
13
>>> square(square(3))
81
```

Existing functions can be used in creating new functions.

```
>>> def sum_of_squares(x, y):
...     return square(x) + square(y)
...
>>> sum_of_squares(2, 3)
13
```

Functions are just like other values, they can assigned, passed as arguments to other functions etc.

```
>>> f = square
>>> f(4)
16

>>> def fxy(f, x, y):
...     return f(x) + f(y)
...
>>> fxy(square, 2, 3)
13
```

It is important to understand, the scope of the variables used in functions.

Lets look at an example.

```
x = 0
y = 0
def incr(x):
    y = x + 1
    return y
incr(5)
print x, y    output >> 0 0
```

Variables assigned in a function, including the arguments are called the local variables to the function. The variables defined in the top-level are called global variables.

Changing the values of `x` and `y` inside the function `incr` won't effect the values of global `x` and `y`.

But, we can use the values of the global variables.

```
pi = 3.14
def area(r):
    return pi * r * r
```

When Python sees use of a variable not defined locally, it tries to find a global variable with that name.

However, you have to explicitly declare a variable as `global` to modify it.

```
numcalls = 0
def square(x):
    global numcalls
    numcalls = numcalls + 1
    return x * x
```

Problem 7: How many multiplications are performed when each of the following lines of code is executed?

```
print square(5)
print square(2*5)
```

Problem 8: What will be the output of the following program?

```
x = 1
def f():
    return x
print x
print f()
```

Problem 9: What will be the output of the following program?

```
x = 1
def f():
    x = 2
    return x
print x
print f()
print x
```

Problem 10: What will be the output of the following program?

```
x = 1
def f():
        y = x
        x = 2
        return x + y
print x
print f()
print x
```

Problem 11: What will be the output of the following program?

```
x = 2
def f(a):
    x = a * a
    return x
y = f(3)
print x, y
```

Functions can be called with keyword arguments.

```
>>> def difference(x, y):
...     return x - y
...
>>> difference(5, 2)
3
>>> difference(x=5, y=2)
3
>>> difference(5, y=2)
3
>>> difference(y=2, x=5)
3
```

And some arguments can have default values.

```
>>> def increment(x, amount=1):
...     return x + amount
...
>>> increment(10)
11
>>> increment(10, 5)
15
>>> increment(10, amount=2)
12
```

There is another way of creating functions, using the `lambda` operator.

```
>>> cube = lambda x: x ** 3
>>> fxy(cube, 2, 3)
35
>>> fxy(lambda x: x ** 3, 2, 3)
35
```

Notice that unlike function defination, lambda doesn't need a `return`. The body of the `lambda` is a single expression.

The `lambda` operator becomes handy when writing small functions to be passed as arguments etc. We'll see more of it as we get into solving more serious problems.

Built-in Functions

Python provides some useful built-in functions.

```
>>> min(2, 3)
2
>>> max(3, 4)
4
```

The built-in function `len` computes length of a string.

```
>>> len("helloworld")
10
```

The built-in function `int` converts string to ingeter and built-in function `str` converts integers and other type of objects to strings.

```
>>> int("50")
50
>>> str(123)
"123"
```

Problem 12: Write a function `count_digits` to find number of digits in the given number.

```
>>> count_digits(5)
1
>>> count_digits(12345)
5
```

Methods

Methods are special kind of functions that work on an object.

For example, `upper` is a method available on string objects.

```
>>> x = "hello"
>>> print x.upper()
HELLO
```

As already mentioned, methods are also functions. They can be assigned to other variables can be called separately.

```
>>> f = x.upper
>>> print f()
HELLO
```

Problem 13: Write a function *istrcmp* to compare two strings, ignoring the case.

```
>>> istrcmp('python', 'Python')
True
>>> istrcmp('LaTeX', 'Latex')
True
>>> istrcmp('a', 'b')
False
```

Conditional Expressions

Python provides various operators for comparing values. The result of a comparison is a boolean value, either `True` or `False`.

```
>>> 2 < 3
False
>>> 2 > 3
True
```

Here is the list of available conditional operators.

- == equal to
- != not equal to
- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to

It is even possible to combine these operators.

```
>>> x = 5
>>> 2 < x < 10
True
>>> 2 < 3 < 4 < 5 < 6
True
```

The conditional operators work even on strings - the ordering being the lexical order.

```
>>> "python" > "perl"
True
>>> "python" > "java"
True
```

There are few logical operators to combine boolean values.

- `a and b` is `True` only if both `a` and `b` are True.
- `a or b` is True if either `a` or `b` is True.
- `not a` is True only if `a` is False.

```
>>> True and True
True
>>> True and False
False
>>> 2 < 3 and 5 < 4
```

```
False
>>> 2 < 3 or 5 < 4
True
```

Problem 14: What will be output of the following program?

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

Problem 15: What will be output of the following program?

```
x = 4
y = 5
p = x < y or x < z
print p
```

Problem 16: What will be output of the following program?

```
True, False = False, True
print True, False
print 2 < 3
```

The if statement

The `if` statement is used to execute a piece of code only when a boolean expression is true.

```
>>> x = 42
>>> if x % 2 == 0: print 'even'
even
>>>
```

In this example, `print 'even'` is executed only when `x % 2 == 0` is `True`.

The code associated with `if` can be written as a separate indented block of code, which is often the case when there is more than one statement to be executed.

```
>>> if x % 2 == 0:
...     print 'even'
...
even
>>>
```

The `if` statement can have optional `else` clause, which is executed when the boolean expression is `False`.

```
>>> x = 3
>>> if x % 2 == 0:
...     print 'even'
... else:
...     print 'odd'
...
odd
>>>
```

The `if` statement can have optional `elif` clauses when there are more conditions to be checked. The `elif` keyword is short for `else if`, and is useful to avoid excessive indentation.

```
>>> x = 42
>>> if x < 10:
...         print 'one digit number'
... elif x < 100:
...     print 'two digit number'
... else:
```

```
...     print 'big number'
...
two digit number
>>>
```

Problem 17: What happens when the following code is executed? Will it give any error? Explain the reasons.

```
x = 2
if x == 2:
    print x
else:
    print y
```

Problem 18: What happens the following code is executed? Will it give any error? Explain the reasons.

```
x = 2
if x == 2:
    print x
else:
    x +
```

Lists

Lists are one of the great datastructures in Python. We are going to learn a little bit about lists now. Basic knowledge of lists is requrired to be able to solve some problems that we want to solve in this chapter.

Here is a list of numbers.

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

And here is a list of strings.

```
>>> x = ["hello", "world"]
```

List can be heterogeneous. Here is a list containings integers, strings and another list.

```
>>> x = [1, 2, "hello", "world", ["another", "list"]]
```

The built-in function `len` works for lists as well.

```
>>> x = [1, 2, 3]
>>> len(x)
3
```

The `[]` operator is used to access individual elements of a list.

```
>>> x = [1, 2, 3]
>>> x[1]
2
>>> x[1] = 4
>>> x[1]
4
```

The first element is indexed with `0`, second with `1` and so on.

We'll learn more about lists in the next chapter.

Modules

Modules are libraries in Python. Python ships with many standard library modules.

A module can be imported using the `import` statement.

Lets look at `time` module for example:

```
>>> import time
>>> time.asctime()
'Tue Sep 11 21:42:06 2012'
```

The `asctime` function from the `time` module returns the current time of the system as a string.

The `sys` module provides access to the list of arguments passed to the program, among the other things.

The `sys.argv` variable contains the list of arguments passed to the program. As a convention, the first element of that list is the name of the program.

Lets look at the following program `echo.py` that prints the first argument passed to it.

```
import sys
print sys.argv[1]
```

Lets try running it.

```
$ python echo.py hello
hello
$ python echo.py hello world
hello
```

There are many more interesting modules in the standard library. We'll learn more about them in the coming chapters.

Problem 19: Write a program `add.py` that takes 2 numbers as command line arguments and prints its sum.

```
$ python add.py 3 5
8
$ python add.py 2 9
11
```

Working with Data

Lists

We've already seen quick introduction to lists in the previous chapter.

```
>>> [1, 2, 3, 4]
[1, 2, 3, 4]
>>> ["hello", "world"]
["hello", "world"]
>>> [0, 1.5, "hello"]
[0, 1.5, "hello"]
>>> [0, 1.5, "hello"]
[0, 1.5, "hello"]
```

A List can contain another list as member.

```
>>> a = [1, 2]
>>> b = [1.5, 2, a]
>>> b
[1.5, 2, [1, 2]]
```

The built-in function `range` can be used to create a list of integers.

```
>>> range(4)
[0, 1, 2, 3]
>>> range(3, 6)
[3, 4, 5]
>>> range(2, 10, 3)
[2, 5, 8]
```

The built-in function `len` can be used to find the length of a list.

```
>>> a = [1, 2, 3, 4]
>>> len(a)
4
```

The + and * operators work even on lists.

```
>>> a = [1, 2, 3]
>>> b = [4, 5]
>>> a + b
[1, 2, 3, 4, 5]
>>> b * 3
[4, 5, 4, 5, 4, 5]
```

This statement increase the number of list while copy the elements of the original list

List can be indexed to get individual entries. Value of index can go from 0 to (length of list - 1).

```
>>> x = [1, 2]
>>> x[0]
1
>>> x[1]
2
```

When a wrong index is used, python gives an error.

```
>>> x = [1, 2, 3, 4]
>>> x[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Negative indices can be used to index the list from right.

```
>>> x = [1, 2, 3, 4]
>>> x[-1]
4
>>> x [-2]
3
```

We can use list slicing to get part of a list.

```
>>> x = [1, 2, 3, 4]
>>> x[0:2]
[1, 2]
>>> x[1:4]
[2, 3, 4]
```

Not included

included

Even negative indices can be used in slicing. For example, the following examples strips the last element from the list.

```
>>> x[0:-1]
[1, 2, 3]
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the list being sliced.

```
>>> x = [1, 2, 3, 4]
>>> a[:2]
[1, 2]
>>> a[2:]
[3, 4]
>>> a[:]
[1, 2, 3, 4]
```

An optional third index can be used to specify the increment, which defaults to 1.

```
>>> x = range(10)
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[0:6:2]
[0, 2, 4]
```

We can reverse a list, just by providing -1 for increment.

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

List members can be modified by assignment.

```
>>> x = [1, 2, 3, 4]
>>> x[1] = 5
>>> x
[1, 5, 3, 4]
```

Presence of a key in a list can be tested using `in` operator.

```
>>> x = [1, 2, 3, 4]
>>> 2 in x
True
>>> 10 in x
False
```

Values can be appended to a list by calling `append` method on list. A method is just like a function, but it is associated with an object and can access that object when it is called. We will learn more about methods when we study classes.

```
>>> a = [1, 2]
>>> a.append(3)
>>> a
[1, 2, 3]
```

value to add at the end of the list

Problem 20: What will be the output of the following program?

```
x = [0, 1, [2]]
x[2][0] = 3
print x
x[2].append(4)
print x
x[2] = 2
print x
```

The for Statement

Python provides `for` statement to iterate over a list. A `for` statement executes the specified block of code for every element in a list.

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

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

The built-in function `zip` takes two lists and returns list of pairs.

```
>>> zip(["a", "b", "c"], [1, 2, 3])
[('a', 1), ('b', 2), ('c', 3)]
```

It is handy when we want to iterate over two lists together.

```
names = ["a", "b", "c"]
values = [1, 2, 3]
for name, value in zip(names, values):
    print name, value
```

Problem 21: Python has a built-in function `sum` to find sum of all elements of a list. Provide an implementation for `sum`.

```
>>> sum([1, 2, 3])
>>> 6
```

Problem 22: What happens when the above `sum` function is called with a list of strings? Can you make your `sum` function work for a list of strings as well.

```
>>> sum(["hello", "world"])
"helloworld"
>>> sum(["aa", "bb", "cc"])
"aabbcc"
```

Problem 23: Implement a function `product`, to compute product of a list of numbers.

```
>>> product([1, 2, 3])
6
```

Problem 24: Write a function `factorial` to compute factorial of a number. Can you use the `product` function defined in the previous example to compute factorial?

```
>>> factorial(4)
24
```

Problem 25: Write a function `reverse` to reverse a list. Can you do this without using list slicing?

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

Problem 26: Python has built-in functions `min` and `max` to compute minimum and maximum of a given list. Provide an implementation for these functions. What happens when you call your `min` and `max` functions with a list of strings?

Problem 27: Cumulative sum of a list `[a, b, c, ...]` is defined as `[a, a+b, a+b+c, ...]`. Write a function `cumulative_sum` to compute cumulative sum of a list. Does your implementation work for a list of strings?

```
>>> cumulative_sum([1, 2, 3, 4])
[1, 3, 6, 10]
>>> cumulative_sum([4, 3, 2, 1])
[4, 7, 9, 10]
```

Problem 28: Write a function `cumulative_product` to compute cumulative product of a list of numbers.

```
>>> cumulative_product([1, 2, 3, 4])
[1, 2, 6, 24]
>>> cumulative_product([4, 3, 2, 1])
[4, 12, 24, 24]
```

Problem 29: Write a function *unique* to find all the unique elements of a list.

```
>>> unique([1, 2, 1, 3, 2, 5])
[1, 2, 3, 5]
```

Problem 30: Write a function *dups* to find all duplicates in the list.

```
>>> dups([1, 2, 1, 3, 2, 5])
[1, 2]
```

Problem 31: Write a function *group(list, size)* that take a list and splits into smaller lists of given size.

```
>>> group([1, 2, 3, 4, 5, 6, 7, 8, 9], 3)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> group([1, 2, 3, 4, 5, 6, 7, 8, 9], 4)
[[1, 2, 3, 4], [5, 6, 7, 8], [9]]
```

Sorting Lists

The `sort` method sorts a list in place.

```
>>> a = [2, 10, 4, 3, 7]
>>> a.sort()
>>> a
[2, 3, 4, 7 10]
```

The built-in function `sorted` returns a new sorted list without modifying the source list.

```
>>> a = [4, 3, 5, 9, 2]
>>> sorted(a)
[2, 3, 4, 5, 9]
>>> a
[4, 3, 5, 9, 2]
```

The behavior of `sort` method and `sorted` function is exactly same except that sorted returns a new list instead of modifying the given list.

The `sort` method works even when the list has different types of objects and even lists.

```
>>> a = ["hello", 1, "world", 45, 2]
>>> a.sort()
>>> a
[1, 2, 45, 'hello', 'world']
>>> a = [[2, 3], [1, 6]]
>>> a.sort()
>>> a
[[1, 6], [2, 3]]
```

We can optionally specify a function as sort key.

```
>>> a = [[2, 3], [4, 6], [6, 1]]
>>> a.sort(key=lambda x: x[1])
>>> a
[[6, 1], [2, 3],  [4 6]]
```

This sorts all the elements of the list based on the value of second element of each entry.

Problem 32: Write a function `lensort` to sort a list of strings based on length.

```
>>> lensort(['python', 'perl', 'java', 'c', 'haskell', 'ruby'])
['c', 'perl', 'java', 'ruby', 'python', 'haskell']
```

Problem 33: Improve the *unique* function written in previous problems to take an optional *key* function as argument and use the return value of the key function to check for uniqueness.

```
>>> unique(["python", "java", "Python", "Java"], key=lambda s: s.lower())
["python", "java"]
```

Tuples

Tuple is a sequence type just like `list`, but it is immutable. A tuple consists of a number of values separated by commas. unable to change

```
>>> a = (1, 2, 3)
>>> a[0]
1
```

The enclosing braces are optional.

```
>>> a = 1, 2, 3
>>> a[0]
1
```

The built-in function `len` and slicing works on tuples too.

```
>>> len(a)
3
>>> a[1:]
2, 3
```

Since parenthesis are also used for grouping, tuples with a single value are represented with an additional comma.

```
>>> a = (1)
>> a
1
>>> b = (1,)
>>> b
(1,)
>>> b[0]
1
```

Sets

Sets are unordered collection of unique elements.

```
>>> x = set([3, 1, 2, 1])
set([1, 2, 3])
```

Python 2.7 introduced a new way of writing sets.

```
>>> x = {3, 1, 2, 1}
set([1, 2, 3])
```

No need to write function set version

New elements can be added to a set using the `add` method.

```
>>> x = set([1, 2, 3])
>>> x.add(4)
>>> x
set([1, 2, 3, 4])
```

Just like lists, the existance of an element can be checked using the `in` operator. However, this operation is faster in sets compared to lists.

```
>>> x = set([1, 2, 3])
>>> 1 in x
True
>>> 5 in x
False
```

Problem 34: Reimplement the *unique* function implemented in the earlier examples using sets.

Strings

Strings also behave like lists in many ways. Length of a string can be found using built-in function `len`.

```
>>> len("abrakadabra")
11
```

Indexing and slicing on strings behave similar to that of lists.

```
>>> a = "helloworld"
>>> a[1]
'e'
>>> a[-2]
'l'
>>> a[1:5]
"ello"
>>> a[:5]
"hello"
>>> a[5:]
"world"
>>> a[-2:]
'ld'
>>> a[:-2]
'hellowor'
>>> a[::-1]
'dlrowolleh'
```

The `in` operator can be used to check if a string is present in another string.

```
>>> 'hell' in 'hello'
True
>>> 'full' in 'hello'
False
>>> 'el' in 'hello'
True
```

There are many useful methods on strings.

The `split` method splits a string using a delimiter. If no delimiter is specified, it uses any whitespace char as delimiter.

```
>>> "hello world".split()
['hello', 'world']
>>> "a,b,c".split(',')
['a', 'b', 'c']
```

The `join` method joins a list of strings.

```
>>> " ".join(['hello', 'world'])
'hello world'
>>> ','.join(['a', 'b', 'c'])
```

The `strip` method returns a copy of the given string with leading and trailing whitespace removed. Optionally a string can be passed as argument to remove characters from that string instead of whitespace.

```
>>> ' hello world\n'.strip()
'hello world'
>>> 'abcdefgh'.strip('abdh')
'cdefg'
```

Python supports formatting values into strings. Although this can include very complicated expressions, the most basic usage is to insert values into a string with the %s placeholder.

```
>>> a = 'hello'
>>> b = 'python'
```

```
>>> "%s %s" % (a, b)
'hello python'
>>> 'Chapter %d: %s' % (2, 'Data Structures')
'Chapter 2: Data Structures'
```

Problem 35: Write a function `extsort` to sort a list of files based on extension.

```
>>> extsort(['a.c', 'a.py', 'b.py', 'bar.txt', 'foo.txt', 'x.c'])
['a.c', 'x.c', 'a.py', 'b.py', 'bar.txt', 'foo.txt']
```

Working With Files

Python provides a built-in function `open` to open a file, which returns a file object.

```
f = open('foo.txt', 'r') # open a file in read mode
f = open('foo.txt', 'w') # open a file in write mode
f = open('foo.txt', 'a') # open a file in append mode
```

The second argument to `open` is optional, which defaults to `'r'` when not specified.

Unix does not distinguish binary files from text files but windows does. On windows `'rb'`, `'wb'`, `'ab'` should be used to open a binary file in read, write and append mode respectively.

Easiest way to read contents of a file is by using the `read` method.

```
>>> open('foo.txt').read()
'first line\nsecond line\nlast line\n'
```

Contents of a file can be read line-wise using `readline` and `readlines` methods. The `readline` method returns empty string when there is nothing more to read in a file.

```
>>> open('foo.txt').readlines()
['first line\n', 'second line\n', 'last line\n']
>>> f = open('foo.txt')
>>> f.readline()
'first line\n'
>>> f.readline()
'second line\n'
>>> f.readline()
'last line\n'
>>> f.readline()
''
```

The `write` method is used to write data to a file opened in write or append mode.

```
>>> f = open('foo.txt', 'w')
>>> f.write('a\nb\nc')
>>> f.close()

>>> f.open('foo.txt', 'a')
>>> f.write('d\n')
>>> f.close()
```

The `writelines` method is convenient to use when the data is available as a list of lines.

```
>>> f = open('foo.txt')
>>> f.writelines(['a\n', 'b\n', 'c\n'])
>>> f.close()
```

Example: Word Count

Lets try to compute the number of characters, words and lines in a file.

Number of characters in a file is same as the length of its contents.

```
def charcount(filename):
    return len(open(filename).read())
```

Number of words in a file can be found by splitting the contents of the file.

```
def wordcount(filename):
    return len(open(filename).read().split())
```

Number of lines in a file can be found from `readlines` method.

```
def linecount(filename):
    return len(open(filename).readlines())
```

Problem 36: Write a program `reverse.py` to print lines of a file in reverse order.

```
$ cat she.txt
She sells seashells on the seashore;
The shells that she sells are seashells I'm sure.
So if she sells seashells on the seashore,
I'm sure that the shells are seashore shells.

$ python reverse.py she.txt
I'm sure that the shells are seashore shells.
So if she sells seashells on the seashore,
The shells that she sells are seashells I'm sure.
She sells seashells on the seashore;
```

Problem 37: Write a program to print each line of a file in reverse order.

Problem 38: Implement unix commands `head` and `tail`. The `head` and `tail` commands take a file as argument and prints its first and last 10 lines of the file respectively.

Problem 39: Implement unix command `grep`. The `grep` command takes a string and a file as arguments and prints all lines in the file which contain the specified string.

```
$ python grep.py she.txt sure
The shells that she sells are seashells I'm sure.
I'm sure that the shells are seashore shells.
```

Problem 40: Write a program *wrap.py* that takes filename and width as aruguments and wraps the lines longer than *width*.

```
$ python wrap.py she.txt 30
I'm sure that the shells are s
eashore shells.
So if she sells seashells on t
he seashore,
The shells that she sells are
seashells I'm sure.
She sells seashells on the sea
shore;
```

Problem 41: The above wrap program is not so nice because it is breaking the line at middle of any word. Can you write a new program *wordwrap.py* that works like *wrap.py*, but breaks the line only at the word boundaries?

```
$ python wordwrap.py she.txt 30
I'm sure that the shells are
seashore shells.
So if she sells seashells on
the seashore,
The shells that she sells are
seashells I'm sure.
She sells seashells on the
seashore;
```

Problem 42: Write a program *center_align.py* to center align all lines in the given file.

```
$ python center_align.py she.txt
  I'm sure that the shells are seashore shells.
    So if she sells seashells on the seashore,
The shells that she sells are seashells I'm sure.
       She sells seashells on the seashore;
```

List Comprehensions

List Comprehensions provide a concise way of creating lists. Many times a complex task can be modelled in a single line.

Here are some simple examples for transforming a list.

```
>>> a = range(10)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [x for x in a]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [x*x for x in a]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x+1 for x in a]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

It is also possible to filter a list using `if` inside a list comprehension.

```
>>> a = range(10)
>>> [x for x in a if x % 2 == 0]
[0, 2, 4, 6, 8]
>>> [x*x for x in a if x%2 == 0]
[0, 4, 8, 36, 64]
```

It is possible to iterate over multiple lists using the built-in function `zip`.

```
>>> a = [1, 2, 3, 4]
>>> b = [2, 3, 5, 7]
>>> zip(a, b)
[(1, 2), (2, 3), (3, 5), (4, 7)]
>>> [x+y for x, y in zip(a, b)]
[3, 5, 8, 11]
```

we can use multiple `for` clauses in single list comprehension.

```
>>> [(x, y) for x in range(5) for y in range(5) if (x+y)%2 == 0]
[(0, 0), (0, 2), (0, 4), (1, 1), (1, 3), (2, 0), (2, 2), (2, 4), (3, 1), (3, 3), (4, 0), (4, 2),

>>> [(x, y) for x in range(5) for y in range(5) if (x+y)%2 == 0 and x != y]
[(0, 2), (0, 4), (1, 3), (2, 0), (2, 4), (3, 1), (4, 0), (4, 2)]

>>> [(x, y) for x in range(5) for y in range(x) if (x+y)%2 == 0]
[(2, 0), (3, 1), (4, 0), (4, 2)]
```

The following example finds all Pythagorean triplets using numbers below 25. `(x, y, z)` is a called pythagorean triplet if `x*x + y*y == z*z`.

```
>>> n = 25
>>> [(x, y, z) for x in range(1, n) for y in range(x, n) for z in range(y, n) if x*x + y*y == z*z
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15), (12, 16, 20)]
```

Problem 43: Provide an implementation for `zip` function using list comprehensions.

```
>>> zip([1, 2, 3], ["a", "b", "c"])
[(1, "a"), (2, "b"), (3, "c")]
```

Problem 44: Python provides a built-in function `map` that applies a function to each element of a list. Provide an implementation for `map` using list comprehensions.

```
>>> def square(x): return x * x
...
>>> map(square, range(5))
[0, 1, 4, 9, 16]
```

Problem 45: Python provides a built-in function `filter(f, a)` that returns items of the list `a` for which `f(item)` returns true. Provide an implementation for `filter` using list comprehensions.

```
>>> def even(x): return x %2 == 0
...
>>> filter(even, range(10))
[0, 2, 4, 6, 8]
```

Problem 46: Write a function `triplets` that takes a number `n` as argument and returns a list of triplets such that sum of first two elements of the triplet equals the third element using numbers below n. Please note that `(a, b, c)` and `(b, a, c)` represent same triplet.

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

Problem 47: Write a function `enumerate` that takes a list and returns a list of tuples containing `(index,item)` for each item in the list.

```
>>> enumerate(["a", "b", "c"])
[(0, "a"), (1, "b"), (2, "c")]
>>> for index, value in enumerate(["a", "b", "c"]):
...     print index, value
0 a
1 b
2 c
```

Problem 48: Write a function `array` to create an 2-dimensional array. The function should take both dimensions as arguments. Value of each element can be initialized to None:

```
>>> a = array(2, 3)
>>> a
[[None, None, None], [None, None, None]]
>>> a[0][0] = 5
[[5, None, None], [None, None, None]]
```

Problem 49: Write a python function `parse_csv` to parse csv (comma separated values) files.

```
>>> print open('a.csv').read()
a,b,c
1,2,3
2,3,4
3,4,5
>>> parse_csv('a.csv')
[['a', 'b', 'c'], ['1', '2', '3'], ['2', '3', '4'], ['3', '4', '5']]
```

Problem 50: Generalize the above implementation of csv parser to support any delimiter and comments.

```
>>> print open('a.txt').read()
# elements are separated by ! and comment indicator is #
a!b!c
1!2!3
2!3!4
3!4!5
>>> parse('a.txt', '!', '#')
[['a', 'b', 'c'], ['1', '2', '3'], ['2', '3', '4'], ['3', '4', '5']]
```

Problem 51: Write a function `mutate` to compute all words generated by a single mutation on a given word. A mutation is defined as inserting a character, deleting a character, replacing a character, or swapping 2 consecutive characters in a string. For simplicity consider only letters from `a` to `z`.

```
>>> words = mutate('hello')
>>> 'helo' in words
True
>>> 'cello' in words
True
>>> 'helol' in words
True
```

Problem 52: Write a function `nearly_equal` to test whether two strings are nearly equal. Two strings `a` and `b` are nearly equal when `a` can be generated by a single mutation on `b`.

```
>>> nearly_equal('python', 'perl')
False
>>> nearly_equal('perl', 'pearl')
True
>>> nearly_equal('python', 'jython')
True
>>> nearly_equal('man', 'woman')
False
```

Dictionaries

Dictionaries are like lists, but they can be indexed with non integer keys also. Unlike lists, dictionaries are not ordered.

```
>>> a = {'x': 1, 'y': 2, 'z': 3}
>>> a['x']
1
>>> a['z']
3
>>> b = {}
>>> b['x'] = 2
>>> b[2] = 'foo'
>>> b[(1, 2)] = 3
>>> b
{(1, 2): 3, 'x': 2, 2: 'foo'}
```

The `del` keyword can be used to delete an item from a dictionary.

```
>>> a = {'x': 1, 'y': 2, 'z': 3}
>>> del a['x']
>>> a
{'y': 2, 'z': 3}
```

The `keys` method returns all keys in a dictionary, the `values` method returns all values in a dictionary and `items` method returns all key-value pairs in a dictionary.

```
>>> a.keys()
['x', 'y', 'z']
>>> a.values()
[1, 2, 3]
>>> a.items()
[('x', 1), ('y', 2), ('z', 3)]
```

The `for` statement can be used to iterate over a dictionary.

```
>>> for key in a: print key
...
x
```

```
y
z
>>> for key, value in a.items(): print key, value
...
x 1
y 2
z 3
```

Presence of a key in a dictionary can be tested using `in` operator or `has_key` method.

```
>>> 'x' in a
True
>>> 'p' in a
False
>>> a.has_key('x')
True
>>> a.has_key('p')
False
```

Other useful methods on dictionaries are `get` and `setdefault`.

```
>>> d = {'x': 1, 'y': 2, 'z': 3}
>>> d.get('x', 5)
1
>>> d.get('p', 5)
5
>>> d.setdefault('x', 0)
1
>>> d
{'x': 1, 'y': 2, 'z': 3}
>>> d.setdefault('p', 0)
0
>>> d
{'y': 2, 'x': 1, 'z': 3, 'p': 0}
```

Dictionaries can be used in string formatting to specify named parameters.

```
>>> 'hello %(name)s' % {'name': 'python'}
'hello python'
>>> 'Chapter %(index)d: %(name)s' % {'index': 2, 'name': 'Data Structures'}
'Chapter 2: Data Structures'
```

Example: Word Frequency

Suppose we want to find number of occurrences of each word in a file. Dictionary can be used to store the number of occurrences for each word.

Lets first write a function to count frequency of words, given a list of words.

```
def word_frequency(words):
    """Returns frequency of each word given a list of words.

        >>> word_frequency(['a', 'b', 'a'])
        {'a': 2, 'b': 1}
    """
    frequency = {}
    for w in words:
        frequency[w] = frequency.get(w, 0) + 1
    return frequency
```

Getting words from a file is very trivial.

```
def read_words(filename):
    return open(filename).read().split()
```

We can combine these two functions to find frequency of all words in a file.

```
def main(filename):
    frequency = word_frequency(read_words(filename))
    for word, count in frequency.items():
        print word, count

if __name__ == "__main__":
    import sys
    main(sys.argv[1])
```

Problem 53: Improve the above program to print the words in the descending order of the number of occurrences.

Problem 54: Write a program to count frequency of characters in a given file. Can you use character frequency to tell whether the given file is a Python program file, C program file or a text file?

Problem 55: Write a program to find anagrams in a given list of words. Two words are called anagrams if one word can be formed by rearranging letters of another. For example 'eat', 'ate' and 'tea' are anagrams.

```
>>> anagrams(['eat', 'ate', 'done', 'tea', 'soup', 'node'])
[['eat', 'ate', 'tea], ['done', 'node'], ['soup']]
```

Problem 56: Write a function `valuesort` to sort values of a dictionary based on the key.

```
>>> valuesort({'x': 1, 'y': 2, 'a': 3})
[3, 1, 2]
```

Problem 57: Write a function `invertdict` to interchange keys and values in a dictionary. For simplicity, assume that all values are unique.

```
>>> invertdict({'x': 1, 'y': 2, 'z': 3})
{1: 'x', 2: 'y', 3: 'z'}
```

Understanding Python Execution Environment

Python stores the variables we use as a dictionary. The `globals()` function returns all the globals variables in the current environment.

```
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None}
>>> x = 1
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None, 'x':
>>> x = 2
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None, 'x':
>>> globals()['x'] = 3
>>> x
3
```

Just like `globals` python also provides a function `locals` which gives all the local variables in a function.

```
>>> def f(a, b): print locals()
...
>>> f(1, 2)
{'a': 1, 'b': 2}
```

One more example:

```
>>> def f(name):
...     return "Hello %(name)s!" % locals()
...
>>> f("Guido")
Hello Guido!
```

Further Reading:

- The article A Plan for Spam by Paul Graham describes a method of detecting spam using probability of occurrence of a word in spam.

Modules

Modules are reusable libraries of code in Python. Python comes with many standard library modules.

A module is imported using the *import* statement.

```
>>> import time
>>> print time.asctime()
'Fri Mar 30 12:59:21 2012'
```

In this example, we've imported the *time* module and called the *asctime* function from that module, which returns current time as a string.

There is also another way to use the import statement.

```
>>> from time import asctime
>>> asctime()
'Fri Mar 30 13:01:37 2012'
```

Here were imported just the *asctime* function from the *time* module.

The *pydoc* command provides help on any module or a function.

```
$ pydoc time
Help on module time:

NAME
    time - This module provides various functions to manipulate time values.
...

$ pydoc time.asctime
Help on built-in function asctime in time:

time.asctime = asctime(...)
    asctime([tuple]) -> string
...
```

On Windows, the *pydoc* command is not available. The work-around is to use, the built-in *help* function.

```
>>> help('time')
Help on module time:

NAME
    time - This module provides various functions to manipulate time values.
...
```

Writing our own modules is very simple.

For example, create a file called *num.py* with the following content.

```
def square(x):
    return x * x
```

```
def cube(x):
    return x * x * x
```

Now open Python interterter:

```
>>> import num
>>> num.square(3)
9
>>> num.cube(3)
27
```

Thats all we've written a python library.

Try *pydoc num* (*pydoc.bat numbers* on Windows) to see documentation for this numbers modules. It won't have any documentation as we haven't providied anything yet.

In Python, it is possible to associate documentation for each module, function using docstrings. Docstrings are strings written at the top of the module or at the beginning of a function.

Lets try to document our *num* module by changing the contents of *num.py*

```
"""The num module provides utilties to work on numbers.

Current it provides square and cube.
"""

def square(x):
    """Computes square of a number."""
    return x * x

def cube(x):
    """Computes cube of a number."""
    return x * x
```

The pydoc command will now show us the doumentation nicely formatted.

```
Help on module num:

NAME
    num - The num module provides utilties to work on numbers.

FILE
    /Users/anand/num.py

DESCRIPTION
    Current it provides square and cube.

FUNCTIONS
    cube(x)
        Computes cube of a number.

    square(x)
        Computes square of a number.
```

Under the hood, python stores the documentation as a special field called *__doc__*.

```
>>> import os
>>> print os.getcwd.__doc__
getcwd() -> path

Return a string representing the current working directory.
```

Standard Library

Python comes with many standard library modules. Lets look at some of the most commonly used ones.

os module

The *os* and *os.path* modules provides functionality to work with files, directories etc.

Problem 58: Write a program to list all files in the given directory.

Problem 59: Write a program *extcount.py* to count number of files for each extension in the given directory. The program should take a directory name as argument and print count and extension for each available file extension.

```
$ python extcount.py src/
14 py
4 txt
1 csv
```

Problem 60: Write a program to list all the files in the given directory along with their length and last modification time. The output should contain one line for each file containing filename, length and modification date separated by tabs. Hint: see help for `os.stat`.

Problem 61: Write a program to print directory tree. The program should take path of a directory as argument and print all the files in it recursively as a tree.

```
$ python dirtree.py foo
foo
|-- a.txt
|-- b.txt
|-- code
|   |-- a.py
|   |-- b.py
|   |-- docs
|   |   |-- a.txt
|   |   \-- b.txt
|   \-- x.py
\-- z.txt
```

urllib module

The *urllib* module provides functionality to download webpages.

```
>>> import urllib
>>> response = urllib.urlopen("http://python.org/")
>>> print response.headers
Date: Fri, 30 Mar 2012 09:24:55 GMT
Server: Apache/2.2.16 (Debian)
Last-Modified: Fri, 30 Mar 2012 08:42:25 GMT
ETag: "105800d-4b7b-4bc71d1db9e40"
Accept-Ranges: bytes
Content-Length: 19323
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

>>> response.header['Content-Type']
'text/html'

>>> content = request.read()
```

Problem 62: Write a program wget.py to download a given URL. The program should accept a URL as argument, download it and save it with the basename of the URL. If the URL ends with a /, consider the basename as index.html.

```
$ python wget.py http://docs.python.org/tutorial/interpreter.html
saving http://docs.python.org/tutorial/interpreter.html as interpreter.html.

$ python wget.py http://docs.python.org/tutorial/
saving http://docs.python.org/tutorial/ as index.html.
```

re module

Problem 63: Write a program antihtml.py that takes a URL as argument, downloads the html from web and print it after stripping html tags.

```
$ python antihtml.py index.html
...
The Python interpreter is usually installed as /usr/local/bin/python on
those machines where it is available; putting /usr/local/bin in your
...
```

Problem 64: Write a function *make_slug* that takes a name converts it into a slug. A slug is a string where spaces and special characters are replaced by a hyphen, typically used to create blog post URL from post title. It should also make sure there are no more than one hyphen in any place and there are no hyphens at the biginning and end of the slug.

```
>>> make_slug("hello world")
'hello-world'
>>> make_slug("hello  world!")
'hello-world'
>>> make_slug(" --hello-  world--")
'hello-world'
```

Problem 65: Write a program *links.py* that takes URL of a webpage as argument and prints all the URLs linked from that webpage.

Problem 66: Write a regular expression to validate a phone number.

json module

Problem 67: Write a program myip.py to print the external IP address of the machine. Use the response from `http://httpbin.org/get` and read the IP address from there. The program should print only the IP address and nothing else.

zipfile module

The *zipfile* module provides interface to read and write zip files.

Here are some examples to demonstate the power of zipfile module.

The following example prints names of all the files in a zip archive.

```
import zipfile
z = zipfile.ZipFile("a.zip")
for name in z.namelist():
    print name
```

The following example prints each file in the zip archive.

```
import zipfile
z = zipfile.ZipFile("a.zip")
for name in z.namelist():
    print
    print "FILE:", name
    print
    print z.read(name)
```

Problem 68: Write a python program *zip.py* to create a zip file. The program should take name of zip file as first argument and files to add as rest of the arguments.

```
$ python zip.py foo.zip file1.txt file2.txt
```

Problem 69: Write a program *mydoc.py* to implement the functionality of pydoc. The program should take the module name as argument and print documentation for the module and each of the functions defined in that module.

```
$ python mydoc.py os
Help on module os:

DESCRIPTION

os - OS routines for Mac, NT, or Posix depending on what system we're on.
...

FUNCTIONS

getcwd()
    ...
```

Hints:

- The *dir* function to get all entries of a module
- The *inspect.isfunction* function can be used to test if given object is a function
- *x.__doc__* gives the docstring for x.
- The *__import__* function can be used to import a module by name

Installing third-party modules

PyPI, The Python Package Index maintains the list of Python packages available. The third-party module developers usually register at PyPI and uploads their packages there.

The standard way to installing a python module is using *pip* or *easy_install*. Pip is more modern and perferred.

Lets start with installing *easy_install*.

- Download the easy_install install script ez_setup.py.
- Run it using Python.

That will install `easy_install`, the script used to install third-party python packages.

Before installing new packages, lets understand how to manage virtual environments for installing python packages.

Earlier the only way of installing python packages was system wide. When used this way, packages installed for one project can conflict with other and create trouble. So people invented a way to create isolated Python environment to install packages. This tool is called virtualenv.

To install `virtualenv`:

```
$ easy_install virtualenv
```

Installing virtualenv also installs the *pip* command, a better replace for *easy_install*.

Once it is installed, create a new virtual env by running the `virtualenv` command.

```
$ virtualenv testenv
```

Now to switch to that env.

On UNIX/Mac OS X:

```
$ source testenv/bin/activate
```

On Windows:

```
> testenv\Scripts\activate
```

Now the virtualenv *testenv* is activated.

Now all the packages installed will be limited to this virtualenv. Lets try to install a third-party package.

```
$ pip install tablib
```

This installs a third-party library called `tablib`.

The `tablib` library is a small little library to work with tabular data and write csv and Excel files.

Here is a simple example.

```
# create a dataset
data = tablib.Dataset()

# Add rows
data.append(["A", 1])
data.append(["B", 2])
data.append(["C", 3])

# save as csv
with open('test.csv', 'wb') as f:
    f.write(data.csv)

# save as Excel
with open('test.xls', 'wb') as f:
    f.write(data.xls)

# save as Excel 07+
with open('test.xlsx', 'wb') as f:
    f.write(data.xlsx)
```

It is even possible to create multi-sheet excel files.

```
sheet1 = tablib.Dataset()
sheet1.append(["A1", 1])
sheet1.append(["A2", 2])

sheet2 = tablib.Dataset()
sheet2.append(["B1", 1])
sheet2.append(["B2", 2])


book = tablib.Databook([data1, data2])
with open('book.xlsx', 'wb') as f:
    f.write(book.xlsx)
```

Problem 70: Write a program `csv2xls.py` that reads a csv file and exports it as Excel file. The prigram should take two arguments. The name of the csv file to read as first argument and the name of the Excel file to write as the second argument.

Problem 71: Create a new virtualenv and install BeautifulSoup. BeautifulSoup is very good library for parsing HTML. Try using it to extract all HTML links from a webpage.

Read the BeautifulSoup documentation to get started.

Object Oriented Programming

State

Suppose we want to model a bank account with support for `deposit` and `withdraw` operations. One way to do that is by using global state as shown in the following example.

```
balance = 0

def deposit(amount):
    global balance
    balance += amount
    return balance

def withdraw(amount):
    global balance
    balance -= amount
    return balance
```

The above example is good enough only if we want to have just a single account. Things start getting complicated if want to model multiple accounts.

We can solve the problem by making the state local, probably by using a dictionary to store the state.

```
def make_account():
    return {'balance': 0}

def deposit(account, amount):
    account['balance'] += amount
    return account['balance']

def withdraw(account, amount):
    account['balance'] -= amount
    return account['balance']
```

With this it is possible to work with multiple accounts at the same time.

```
>>> a = make_account()
>>> b = make_account()
>>> deposit(a, 100)
100
>>> deposit(b, 50)
50
>>> withdraw(b, 10)
40
>>> withdraw(a, 10)
90
```

Classes and Objects

```
class BankAccount:
    def __init__(self):
        self.balance = 0

    def withdraw(self, amount):
```

```
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance

>>> a = BankAccount()
>>> b = BankAccount()
>>> a.deposit(100)
100
>>> b.deposit(50)
50
>>> b.withdraw(10)
40
>>> a.withdraw(10)
90
```

Inheritance

Let us try to create a little more sophisticated account type where the account holder has to maintain a predetermined minimum balance.

```
class MinimumBalanceAccount(BankAccount):
    def __init__(self, minimum_balance):
        BankAccount.__init__(self)
        self.minimum_balance = minimum_balance

    def withdraw(self, amount):
        if self.balance - amount < self.minimum_balance:
            print 'Sorry, minimum balance must be maintained.'
        else:
            BankAccount.withdraw(self, amount)
```

Problem 72: What will the output of the following program.

```
class A:
    def f(self):
        return self.g()

    def g(self):
        return 'A'

class B(A):
    def g(self):
        return 'B'

a = A()
b = B()
print a.f(), b.f()
print a.g(), b.g()
```

Example: Drawing Shapes

```
class Canvas:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.data = [[' '] * width for i in range(height)]

    def setpixel(self, row, col):
        self.data[row][col] = '*'
```

```
    def getpixel(self, row, col):
        return self.data[row][col]

    def display(self):
        print "\n".join(["".join(row) for row in self.data])

class Shape:
    def paint(self, canvas): pass

class Rectangle(Shape):
    def __init__(self, x, y, w, h):
        self.x = x
        self.y = y
        self.w = w
        self.h = h

    def hline(self, x, y, w):
        pass

    def vline(self, x, y, h):
        pass

    def paint(self, canvas):
        hline(self.x, self.y, self.w)
        hline(self.x, self.y + self.h, self.w)
        vline(self.x, self.y, self.h)
        vline(self.x + self.w, self.y, self.h)

class Square(Rectangle):
    def __init__(self, x, y, size):
        Rectangle.__init__(self, x, y, size, size)

class CompoundShape(Shape):
    def __init__(self, shapes):
        self.shapes = shapes

    def paint(self, canvas):
        for s in self.shapes:
            s.paint(canvas)
```

Special Class Methods

In Python, a class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to operator overloading, allowing classes to define their own behavior with respect to language operators.

For example, the + operator invokes `__add__` method.

```
>>> a, b = 1, 2
>>> a + b
3
>>> a.__add__(b)
3
```

Just like `__add__` is called for + operator, `__sub__`, `__mul__` and `__div__` methods are called for `-`, `*`, and `/` operators.

Example: Rational Numbers

Suppose we want to do arithmetic with rational numbers. We want to be able to add, subtract, multiply, and divide them and to test whether two rational numbers are equal.

We can add, subtract, multiply, divide, and test equality by using the following relations:

```
n1/d1 + n2/d2 = (n1*d2 + n2*d1)/(d1*d2)
n1/d1 - n2/d2 = (n1*d2 - n2*d1)/(d1*d2)
n1/d1 * n2/d2 = (n1*n2)/(d1*d2)
(n1/d1) / (n2/d2) = (n1*d2)/(d1*n2)

n1/d1 == n2/d2 if and only if n1*d2 == n2*d1
```

Lets write the rational number class.

```
class RationalNumber:
    """
    Rational Numbers with support for arthmetic operations.

        >>> a = RationalNumber(1, 2)
        >>> b = RationalNumber(1, 3)
        >>> a + b
        5/6
        >>> a - b
        1/6
        >>> a * b
        1/6
        >>> a/b
        3/2
    """
    def __init__(self, numerator, denominator=1):
        self.n = numerator
        self.d = denominator

    def __add__(self, other):
        if not isinstance(other, RationalNumber):
            other = RationalNumber(other)

        n = self.n * other.d + self.d * other.n
        d = self.d * other.d
        return RationalNumber(n, d)

    def __sub__(self, other):
        if not isinstance(other, RationalNumber):
            other = RationalNumber(other)

        n1, d1 = self.n, self.d
        n2, d2 = other.n, other.d
        return RationalNumber(n1*d2 - n2*d1, d1*d2)

    def __mul__(self, other):
        if not isinstance(other, RationalNumber):
            other = RationalNumber(other)

        n1, d1 = self.n, self.d
        n2, d2 = other.n, other.d
        return RationalNumber(n1*n2, d1*d2)

    def __div__(self, other):
        if not isinstance(other, RationalNumber):
            other = RationalNumber(other)

        n1, d1 = self.n, self.d
        n2, d2 = other.n, other.d
        return RationalNumber(n1*d2, d1*n2)

    def __str__(self):
        return "%s/%s" % (self.n, self.d)
```

```
    __repr__ = __str__
```

Errors and Exceptions

We've already seen exceptions in various places. Python gives `NameError` when we try to use a variable that is not defined.

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

try adding a string to an integer:

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

try dividing a number by 0:

```
>>> 2/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

or, try opening a file that is not there:

```
>>> open("not-there.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'not-there.txt'
```

Python raises exception in case errors. We can write programs to handle such errors. We too can raise exceptions when an error case in encountered.

Exceptions are handled by using the try-except statements.

```
def main():
    filename = sys.argv[1]
    try:
        for row in parse_csv(filename):
            print row
    except IOError:
        print >> sys.stderr, "The given file doesn't exist: ", filename
        sys.exit(1)
```

This above example prints an error message and exits with an error status when an IOError is encountered.

The *except* statement can be written in multiple ways:

```
# catch all exceptions
try:
    ...
except:

# catch just one exception
try:
    ...
except IOError:
    ...

# catch one exception, but provide the exception object
```

```
try:
    ...
except IOError, e:
    ...

# catch more than one exception
try:
    ...
except (IOError, ValueError), e:
    ...
```

It is possible to have more than one *except* statements with one *try*.

```
try:
    ...
except IOError, e:
    print >> sys.stderr, "Unable to open the file (%s): %s" % (str(e), filename)
    sys.exit(1)
except FormatError, e:
    print >> sys.stderr, "File is badly formatted (%s): %s" % (str(e), filename)
```

The *try* statement can have an optional *else* clause, which is executed only if no exception is raised in the try-block.

```
try:
    ...
except IOError, e:
    print >> sys.stderr, "Unable to open the file (%s): %s" % (str(e), filename)
    sys.exit(1)
else:
    print "successfully opened the file", filename
```

There can be an optional *else* clause with a *try* statement, which is executed irrespective of whether or not exception has occured.

```
try:
    ...
except IOError, e:
    print >> sys.stderr, "Unable to open the file (%s): %s" % (str(e), filename)
    sys.exit(1)
finally:
    delete_temp_files()
```

Exception is raised using the raised keyword.

```
raise Exception("error message")
```

All the exceptions are extended from the built-in *Exception* class.

class ParseError(Exception): pass

Problem 73: What will be the output of the following program?

```
try:
    print "a"
except:
    print "b"
else:
    print "c"
finally:
    print "d"
```

Problem 74: What will be the output of the following program?

```
try:
    print "a"
    raise Exception("doom")
```

```
except:
    print "b"
else:
    print "c"
finally:
    print "d"
```

Problem 75: What will be the output of the following program?

```
def f():
    try:
        print "a"
        return
    except:
        print "b"
    else:
        print "c"
    finally:
        print "d"

f()
```

Iterators & Generators

Iterators

We use `for` statement for looping over a list.

```
>>> for i in [1, 2, 3, 4]:
...     print i,
...
1
2
3
4
```

If we use it with a string, it loops over its characters.

```
>>> for c in "python":
...     print c
...
p
y
t
h
o
n
```

If we use it with a dictionary, it loops over its keys.

```
>>> for k in {"x": 1, "y": 2}:
...     print k
...
y
x
```

If we use it with a file, it loops over lines of the file.

```
>>> for line in open("a.txt"):
...     print line,
...
```

```
first line
second line
```

So there are many types of objects which can be used with a for loop. These are called iterable objects.

There are many functions which consume these iterables.

```
>>> ",".join(["a", "b", "c"])
'a,b,c'
>>> ",".join({"x": 1, "y": 2})
'y,x'
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
>>> list({"x": 1, "y": 2})
['y', 'x']
```

The Iteration Protocol

The built-in function `iter` takes an iterable object and returns an iterator.

```
>>> x = iter([1, 2, 3])
>>> x
<listiterator object at 0x1004ca850>
>>> x.next()
1
>>> x.next()
2
>>> x.next()
3
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Each time we call the `next` method on the iterator gives us the next element. If there are no more elements, it raises a *StopIteration*.

Iterators are implemented as classes. Here is an iterator that works like built-in `xrange` function.

```
class yrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

The `__iter__` method is what makes an object iterable. Behind the scenes, the *iter* function calls `__iter__` method on the given object.

The return value of `__iter__` is an iterator. It should have a `next` method and raise `StopIteration` when there are no more elements.

Lets try it out:

```
>>> y = yrange(3)
>>> y.next()
0
>>> y.next()
1
>>> y.next()
2
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in next
StopIteration
```

Many built-in functions accept iterators as arguments.

```
>>> list(yrange(5))
[0, 1, 2, 3, 4]
>>> sum(yrange(5))
10
```

In the above case, both the iterable and iterator are the same object. Notice that the `__iter__` method returned `self`. It need not be the case always.

```
class zrange:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return zrange_iter(self.n)

class zrange_iter:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        # Iterators are iterables too.
        # Adding this functions to make them so.
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

If both iteratable and iterator are the same object, it is consumed in a single iteration.

```
>>> y = yrange(5)
>>> list(y)
[0, 1, 2, 3, 4]
>>> list(y)
[]
>>> z = zrange(5)
>>> list(z)
[0, 1, 2, 3, 4]
>>> list(z)
[0, 1, 2, 3, 4]
```

Problem 76: Write an iterator class `reverse_iter`, that takes a list and iterates it from the reverse direction.
::

```
>>> it = reverse_iter([1, 2, 3, 4])
>>> it.next()
4
>>> it.next()
3
>>> it.next()
2
>>> it.next()
1
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Generators

Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.

```
def yrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Each time the `yield` statement is executed the function generates a new value.

```
>>> y = yrange(3)
>>> y
<generator object yrange at 0x401f30>
>>> y.next()
0
>>> y.next()
1
>>> y.next()
2
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

So a generator is also an iterator. You don't have to worry about the iterator protocol.

The word "generator" is confusingly used to mean both the function that generates and what it generates. In this chapter, I'll use the word "generator" to mean the genearted object and "generator function" to mean the function that generates it.

Can you think about how it is working internally?

When a generator function is called, it returns a generator object without even beginning execution of the function. When `next` method is called for the first time, the function starts executing until it reaches `yield` statement. The yielded value is returned by the `next` call.

The following example demonstrates the interplay between `yield` and call to `next` method on generator object.

```
>>> def foo():
...     print "begin"
...     for i in range(3):
...         print "before yield", i
...         yield i
...         print "after yield", i
...     print "end"
...
```

```
>>> f = foo()
>>> f.next()
begin
before yield 0
0
>>> f.next()
after yield 0
before yield 1
1
>>> f.next()
after yield 1
before yield 2
2
>>> f.next()
after yield 2
end
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Lets see an example:

```
def integers():
    """Infinite sequence of integers."""
    i = 1
    while True:
        yield i
        i = i + 1

def squares():
    for i in integers():
        yield i * i

def take(n, seq):
    """Returns first n values from the given sequence."""
    seq = iter(seq)
    result = []
    try:
        for i in range(n):
            result.append(seq.next())
    except StopIteration:
        pass
    return result

print take(5, squares()) # prints [1, 4, 9, 16, 25]
```

Generator Expressions

Generator Expressions are generator version of list comprehensions. They look like list comprehensions, but returns a generator back instead of a list.

```
>>> a = (x*x for x in range(10))
>>> a
<generator object <genexpr> at 0x401f08>
>>> sum(a)
285
```

We can use the generator expressions as arguments to various functions that consume iterators.

```
>>> sum((x*x for x in range(10)))
285
```

When there is only one argument to the calling function, the parenthesis around generator expression can be omitted.

```
>>> sum(x*x for x in range(10))
285
```

Another fun example:

Lets say we want to find first 10 (or any n) pythogorian triplets. A triplet `(x, y, z)` is called pythogorian triplet if `x*x + y*y == z*z`.

It is easy to solve this problem if we know till what value of z to test for. But we want to find first n pythogorian triplets.

```
>>> pyt = ((x, y, z) for z in integers() for y in xrange(1, z) for x in range(1, y) if x*x + y*y =
>>> take(10, pyt)
[(3, 4, 5), (6, 8, 10), (5, 12, 13), (9, 12, 15), (8, 15, 17), (12, 16, 20), (15, 20, 25), (7, 24,
```

Example: Reading multiple files

Lets say we want to write a program that takes a list of filenames as arguments and prints contents of all those files, like `cat` command in unix.

The traditional way to implement it is:

```
def cat(filenames):
    for f in filenames:
        for line in open(f):
            print line,
```

Now, lets say we want to print only the line which has a particular substring, like `grep` command in unix.

```
def grep(pattern, filenames):
    for f in filenames:
        for line in open(f):
            if pattern in line:
                print line,
```

Both these programs have lot of code in common. It is hard to move the common part to a function. But with generators makes it possible to do it.

```
def readfiles(filenames):
    for f in filenames:
        for line in open(f):
            yield line

def grep(pattern, lines):
    return (line for line in lines if pattern in line)

def printlines(lines):
    for line in lines:
        print line,

def main(pattern, filenames):
    lines = readfiles(filenames)
    lines = grep(pattern, lines)
    printlines(lines)
```

The code is much simpler now with each function doing one small thing. We can move all these functions into a separate module and reuse it in other programs.

Problem 77: Write a program that takes one or more filenames as arguments and prints all the lines which are longer than 40 characters.

Problem 78: Write a function `findfiles` that recursively descends the directory tree for the specified directory and generates paths of all the files in the tree.

Problem 79: Write a function to compute the number of python files (.py extension) in a specified directory recursively.

Problem 80: Write a function to compute the total number of lines of code in all python files in the specified directory recursively.

Problem 81: Write a function to compute the total number of lines of code, ignoring empty and comment lines, in all python files in the specified directory recursively.

Problem 82: Write a program `split.py`, that takes an integer `n` and a filename as command line arguments and splits the file into multiple small files with each having `n` lines.

Itertools

The itertools module in the standard library provides lot of intersting tools to work with iterators.

Lets look at some of the interesting functions.

chain – chains multiple iterators together.

```
>>> it1 = iter([1, 2, 3])
>>> it2 = iter([4, 5, 6])
>>> itertools.chain(it1, it2)
[1, 2, 3, 4, 5, 6]
```

izip – iterable version of zip

```
>>> for x, y in itertools.izip(["a", "b", "c"], [1, 2, 3]):
...     print x, y
...
a 1
b 2
c 3
```

Problem 83: Write a function `peep`, that takes an iterator as argument and returns the first element and an equivalant iterator.

```
>>> it = iter(range(5))
>>> x, it1 = peep(it)
>>> print x, list(it1)
0 [0, 1, 2, 3, 4]
```

Problem 84: The built-in function `enumerate` takes an iteratable and returns an iterator over pairs (index, value) for each value in the source.

```
>>> list(enumerate(["a", "b", "c"])
[(0, "a"), (1, "b"), (2, "c")]
>>> for i, c in enumerate(["a", "b", "c"]):
...     print i, c
...
0 a
1 b
2 c
```

Write a function `my_enumerate` that works like `enumerate`.

Problem 85: Implement a function `izip` that works like `itertools.izip`.

Further Reading

- Generator Tricks For System Programers by David Beazly is an excellent in-depth introduction to generators and generator expressions.

Functional Programming

Recursion

Defining solution of a problem in terms of the same problem, typically of smaller size, is called recursion. Recursion makes it possible to express solution of a problem very concisely and elegantly.

A function is called recursive if it makes call to itself. Typically, a recursive function will have a terminating condition and one or more recursive calls to itself.

Example: Computing Exponent

Mathematically we can define exponent of a number in terms of its smaller power.

```
def exp(x, n):
    """
    Computes the result of x raised to the power of n.

        >>> exp(2, 3)
        8
        >>> exp(3, 2)
        9
    """
    if n == 0:
        return 1
    else:
        return x * exp(x, n-1)
```

Lets look at the execution pattern.

```
exp(2, 4)
+-- 2 * exp(2, 3)
|       +-- 2 * exp(2, 2)
|       |       +-- 2 * exp(2, 1)
|       |       |       +-- 2 * exp(2, 0)
|       |       |       |       +-- 1
|       |       |       +-- 2 * 1
|       |       |       +-- 2
|       |       +-- 2 * 2
|       |       +-- 4
|       +-- 2 * 4
|       +-- 8
+-- 2 * 8
+-- 16
```

Number of calls to the above `exp` function is proportional to size of the problem, which is `n` here.

We can compute exponent in fewer steps if we use successive squaring.

```
def fast_exp(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return fast_exp(x*x, n/2))
    else:
        return x * fast_exp(x, n-1)
```

Lets look at the execution pattern now.

```
fast_exp(2, 10)
+-- fast_exp(4, 5) # 2 * 2
|   +-- 4 * fast_exp(4, 4)
```

```
|   |       +-- fast_exp(16, 2) # 4 * 4
|   |       |   +-- fast_exp(256, 1) # 16 * 16
|   |       |   |   +-- 256 * fast_exp(256, 0)
|   |       |   |               +-- 1
|   |       |   |   +-- 256 * 1
|   |       |   |   +-- 256
|   |       |   +-- 256
|   |       +-- 256
|   +-- 4 * 256
|   +-- 1024
+-- 1024
1024
```

Problem 86: Implement a function `product` to multiply 2 numbers recursively using + and − operators only.

Example: Flatten a list

Supposed you have a nested list and want to flatten it.

```
def flatten_list(a, result=None):
    """Flattens a nested list.

        >>> flatten_list([ [1, 2, [3, 4] ], [5, 6], 7])
        [1, 2, 3, 4, 5, 6, 7]
    """
    if result is None:
        result = []

    for x in a:
        if isinstance(x, list):
            flatten_list(x, result)
        else:
            result.append(x)

    return result
```

Problem 87: Write a function `flatten_dict` to flatten a nested dictionary by joining the keys with `.` character.

```
>>> flatten_dict({'a': 1, 'b': {'x': 2, 'y': 3}, 'c': 4})
{'a': 1, 'b.x': 2, 'b.y': 3, 'c': 4}
```

Problem 88: Write a function `unflatten_dict` to do reverse of `flatten_dict`.

```
>>> unflatten_dict({'a': 1, 'b.x': 2, 'b.y': 3, 'c': 4})
{'a': 1, 'b': {'x': 2, 'y': 3}, 'c': 4}
```

Problem 89: Write a function `treemap` to map a function over nested list.

```
>>> treemap(lambda x: x*x, [1, 2, [3, 4, [5]]])
[1, 4, [9, 16, [25]]]
```

Problem 90: Write a function `tree_reverse` to reverse elements of a nested-list recursively.

```
>>> tree_reverse([[1, 2], [3, [4, 5]], 6])
[6, [[5, 4], 3], [2, 1]]
```

Example: JSON Encode

Lets look at more commonly used example of serializing a python datastructure into JSON (JavaScript Object Notation).

Here is an example of JSON record.

```
{
    "name": "Advanced Python Training",
    "date": "October 13, 2012",
    "completed": false,
    "instructor": {
        "name": "Anand Chitipothu",
        "website": "http://anandology.com/"
    },
    "participants": [
        {
            "name": "Participant 1",
            "email": "email1@example.com"
        },
        {
            "name": "Participant 2",
            "email": "email2@example.com"
        }
    ]
}
```

It looks very much like Python dictionaries and lists. There are some differences though. Strings are always enclosed in double quotes, booleans are represented as `true` and `false`.

The standard library module *json* provides functionality to work in JSON. Lets try to implement it now as it is very good example of use of recursion.

For simplicity, lets assume that strings will not have any special characters and can have space, tab and newline characters.

```
def json_encode(data):
    if isinstance(data, bool):
        if data:
            return "true"
        else:
            return "false"
    elif isinstance(data, (int, float)):
        return str(data)
    elif isinstance(data, str):
        return '"' + escape_string(data) + '"'
    elif isinstance(data, list):
        return "[" + ", ".join(json_encode(d) for d in data) + "]"
    else:
        raise TypeError("%s is not JSON serializable" % repr(data))

def escape_string(s):
    """Escapes double-quote, tab and new line characters in a string."""
    s = s.replace('"', '\\"')
    s = s.replace("\t", "\\t")
    s = s.replace("\n", "\\n")
    return s
```

This handles booleans, integers, strings, floats and lists, but doesn't handle dictionaries yet. That is left an exercise to the readers.

If you notice the block of code that is handling lists, we are calling *json_encode* recursively for each element of the list, that is required because each element can be of any type, even a list or a dictionary.

Problem 91: Complete the above implementation of `json_encode` by handling the case of dictionaries.

Problem 92: Implement a program `dirtree.py` that takes a directory as argument and prints all the files in that directory recursively as a tree. Hint: Use `os.listdir` and `os.path.isdir` funtions.

```
$ python dirtree.py foo/
foo/
|-- a.txt
|-- b.txt
|-- bar/
|   |-- p.txt
|   `-- q.txt
`-- c.txt
```

Problem 93: Write a function `count_change` to count the number of ways to change any given amount. Available coins are also passed as argument to the function.

```
>>> count_change(10, [1, 5])
3
>>> count_change(10, [1, 2])
6
>>> count_change(100, [1, 5, 10, 25, 50])
292
```

Problem 94: Write a function `permute` to compute all possible permutations of elements of a given list.

```
>>> permute([1, 2, 3])
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Higher Order Functions & Decorators

In Python, functions are first-class objects. They can be passed as arguments to other functions and a new functions can be returned from a function call.

Example: Tracing Function Calls

For example, consider the following `fib` function.

```
def fib(n):
    if n is 0 or n is 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Suppose we want to trace all the calls to the `fib` function. We can write a higher order function to return a new function, which prints whenever `fib` function is called.

```
def trace(f):
    def g(x):
        print f.__name__, x
        value = f(x)
        print 'return', repr(value)
        return value
    return g

fib = trace(fib)
print fib(3)
```

This produces the following output.

```
fib 3
fib 2
fib 1
return 1
fib 0
return 1
```

```
return 2
fib 1
return 1
return 3
3
```

Noticed that the trick here is at `fib = trace(fib)`. We have replaced the function `fib` with a new function, so whenever that function is called recursively, it is the our new function, which prints the trace before calling the orginal function.

To make the output more readable, let us indent the function calls.

```
def trace(f):
    f.indent = 0
    def g(x):
        print '|  ' * f.indent + '|--', f.__name__, x
        f.indent += 1
        value = f(x)
        print '|  ' * f.indent + '|--', 'return', repr(value)
        f.indent -= 1
        return value
    return g

fib = trace(fib)
print fib(4)
```

This produces the following output.

```
$ python fib.py
|-- fib 4
|  |-- fib 3
|  |  |-- fib 2
|  |  |  |-- fib 1
|  |  |  |  |-- return 1
|  |  |  |-- fib 0
|  |  |  |  |-- return 1
|  |  |  |-- return 2
|  |  |-- fib 1
|  |  |  |-- return 1
|  |  |-- return 3
|  |-- fib 2
|  |  |-- fib 1
|  |  |  |-- return 1
|  |  |-- fib 0
|  |  |  |-- return 1
|  |  |-- return 2
|  |-- return 5
5
```

This pattern is so useful that python has special syntax for specifying this concisely.

```
@trace
def fib(n):
    ...
```

It is equivalant of adding `fib = trace(fib)` after the function definition.

Example: Memoize

In the above example, it is clear that number of function calls are growing exponentially with the size of input and there is lot of redundant computation that is done.

Suppose we want to get rid of the redundant computation by caching the result of `fib` when it is called for the

first time and reuse it when it is needed next time. Doing this is very popular in functional programming world and it is called `memoize`.

```
def memoize(f):
    cache = {}
    def g(x):
        if x not in cache:
            cache[x] = f(x)
        return cache[x]
    return g

fib = trace(fib)
fib = memoize(fib)
print fib(4)
```

If you notice, after `memoize`, growth of `fib` has become linear.

```
|-- fib 4
|   |-- fib 3
|   |   |-- fib 2
|   |   |   |-- fib 1
|   |   |   |   |-- return 1
|   |   |   |-- fib 0
|   |   |   |   |-- return 1
|   |   |   |-- return 2
|   |   |-- return 3
|   |-- return 5
5
```

Problem 95: Write a function `profile`, which takes a function as argument and returns a new function, which behaves exactly similar to the given function, except that it prints the time consumed in executing it.

```
>>> fib = profile(fib)
>>> fib(20)
time taken: 0.1 sec
10946
```

Problem 96: Write a function `vectorize` which takes a function `f` and return a new function, which takes a list as argument and calls `f` for every element and returns the result as a list.

```
>>> def square(x): return x * x
...
>>> f = vectorize(square)
>>> f([1, 2, 3])
[1, 4, 9]
>>> g = vectorize(len)
>>> g(["hello", "world"])
[5, 5]
>>> g([[1, 2], [2, 3, 4]])
[2, 3]
```

Example: unixcommand decorator

Many unix commands have a typical pattern. They accept multiple filenames as arguments, does some processing and prints the lines back. Some examples of such commands are `cat` and `grep`.

```
def unixcommand(f):
    def g(filenames):
        printlines(out for line in readlines(filenames)
                       for out in f(line))
    return g
```

Lets see how to use it.

```
@unixcommand
def cat(line):
    yield line

@unixcommand
def lowercase(line):
    yield line.lower()
```

exec & eval

Python privides the whole interpreter as a built-in function. You can pass a string and ask it is execute that piece of code at run time.

For example:

```
>>> exec("x = 1")
>>> x
1
```

By default `exec` works in the current environment, so it updated the globals in the above example. It is also possible to specify an environment to `exec`.

```
>>> env = {'a' : 42}
>>> exec('x = a+1', env)
>>> print env['x']
43
```

It is also possible to create functions or classes dynamically using `exec`, though it is usually not a good idea.

```
>>> code = 'def add_%d(x): return x + %d'
>>> for i in range(1, 5):
...     exec(code % (i, i))
...
>>> add_1(3)
4
>>> add_3(3)
6
```

`eval` is like `exec` but it takes an expression and returns its value.

```
>>> eval("2+3")
5
>>> a = 2
>>> eval("a * a")
4
>>> env = {'x' : 42}
>>> eval('x+1', env)
43
```

CHAPTER 3

License