Chapter 1, Boss Statements

John M. Morrison

November 15, 2020

Contents

0 Introduction

Here is what we have at our disposal so far. Python objects can respond to operators and methods. We have a rich ecosystem of very useful objects, including collections such as sets, lists and dictionaries. In this way, a Python method can do quite a bit of work for us.

Despite this, our palette for writing programs is pretty limited. All we can do at this time is to write a list of Python worker statements and to execute them *in seratum*. Since you have programmed before you will find this to be very dull indeed. We shall remedy that. The cure is the boss statement.

Statements work like clauses in English. A statement that reads as a complete sentence, or independent clause, is called a *worker statement*. A worker statement is a simple imperative sentence with tacit subject "Python." Here are some some examples.

1. `x = 5` Read this as "`x` gets 5" or "make `x` point at the value 5."
2. `print(x*x + 3)` Read this "Evaluate the expression `x*x + 3` and put it to `stdout`."
3. `x = 3*x - 7` Read this as "Evaluate the expression `3*x + 7` and have `x` point at the result.

1 Our First Boss Statement: Functions

Functions in Python work much like functions in other languages such as Java, JavaScript, or C. Let us begin by showing a function that squares a number.

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

The first statement `def square(x)` should be read, "To define `square(x)`," Notice that it is a grammatically incomplete sentence. This statement is an example of a *boss statement*, which controls the flow of execution in a program. A boss statement must own a *block* of code, which is one or more lines of code underneath it that are indented the same amount. This block of code, combined with the boss statement, constitutes a grammatically complete sentence. Notice that the block is indented one tab stop.

Note for users of `vi/vim` Change to your home directory now and edit our `.vimrc` file; if you don't have it, create it. Make sure these lines are in it

```
syntax on
set tabstop=4
set et
```

The first line gives you syntax coloring for all of your favorite file types. The second line sets the tab stop at 4 spaces. Do not put spaces around the equal sign or you will get annoying error messages! The third line turns every tab into 4 spaces. This eliminates a serious aggravation. Do this now and there will be

peace in the valley.... or else. This will spare you annoying "inconsistent use of tabs and spaces," and other whitespace-related errors. You can also do this in other text editors by hunting in the preferences.

Now add to your function as follows. Put this in a file named `boss.py`.

```
def square(x):
    return x*x
print(type(square))
print(f"The Square of 10 is {square(10)}")
```

Now we run it and we see this.

```
unix> python boss.py
<class 'function'>
The Square of 10 is  100
```

A function is just another Python object! Now we see why `def square(x):` is really not a complete sentence. It is about as complete as the half-done assignment `x =`. In fact, defining a function *is* an assignment.

On the second line we are using, or *calling* our function. It does what it expected; it squares the number given or *passed* it and it outputs, or returns 100.

The code following the `def` boss statement `return x*x` is its block of code. Read it now; you have the complete sentence, "To define square(x), return x*x." Et voila! The block completes the boss statement grammatically. The block of a boss statement can contain one or more lines of code. The block ends where the indentation ends. Boss statements can be nested; a block of code can include boss statements, each of which owns a block of code.

2 Scoping and How Functions Work

Variables created outside of any function live in the program's *global scope*. It is best not to minimize the use of these, for they will give you worlds of pain and little joy or use. In this book, we will eschew them assiduously.

Functions residing directly in a file also have global scope. This means that functions can be seen by other functions and that they can call each other. This is good. It allows you to create "teams" of functions that work together to accomplish a task.

Let us do something interesting with our little square function. Make this program called evilScope.py

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

print(square(12))
print(x)
print(y)
```

We now get hissed at by an angry Python.

```
unix> python evilScope.py
144
Traceback (most recent call last):
  File "evilScope.py", line 6, in <module>
    print(x)
NameError: name 'x' is not defined
```

It would seem that `x` would be storing the value `12`. Evidently not. Here is what happened.

1. A copy of the memory address stored by the variable is sent to the function's parameter `x`.
2. If a literal is passed, a copy of its location in memory is sent to the parameter `x`.
3. An internal variable `y` got created here. It got the value `144`.
4. The memory address where `144` is stored is passed back to the `print` function, which prints `144`.
5. Once the function returned, all traces of the variable `x` disappeared. The same thing happened to `y`. To see this, delete the line where `x` is printed and re-run the program.

We have just learned two things about Python. One is that Python has *function scope*. Arguments of functions and variables created inside of a functions are not visible outside of that function.

The other is that Python is a pure pass-by-value language. When you pass a variable or a literal to a function, the function actually gets a *copy* of the memory address of that item.

You might ask, "Why all of this hiding of stuff?" You have seen us use some built-in functions such as `len`, `min` and `max`. Would you want to have to worry about variables created inside of these and have to keep track of their names? This would quickly lead to choking levels of complexity. When using functions all we need to care about is what sorts of parameters they need and what they

do. This frees us to think about the problem we are trying to solve and not about the internal finickiness of our tools.

When writing a function, you must concern yourself with these major areas.

1. **parameters** How many of these are there are of what type(s) should they be?
2. **preconditions** More generally, what should be true when you call this function? Really, a description of your parameters is part of the preconditions.
3. **side-effects** Does this function leave stuff behind once it returns? Does it create a file? Does it put things to `stdout`? Does it modify the state of any mutable objects?
4. **return value** What kind of object, if any is returned by the function? If you do not return anything a graveyard object named `None` is automatically returned.
5. **postconditions** This is what is true when a function is done executing. This contains a description of a function's side-effects and return value, if any.

As with any programming language, Python has a system that makes it easy to document your function. You insert a *docstring*, as shown here.

```
def square(x):
    """precondition:  x is a number (integer or float)
postcondition:  This function returns the square of x.
It has no side-effects."""
    y = x*x
    return y
print(square.__doc__)
```

Now we run this; you will see how to print the docstring of any function that has one.

```
unix> python docstring.py
precondition:  x is a number (integer or float)
postcondition:  This function returns the square of x.
It has no side-effects.
```

3 Conditional Logic

A mainstay of computer languages is the `if` statement and its friends. We will now explore these in Python. There are three important boss statements in Python's conditional logic, `if`, `else`, and `elif`.

We being by looking at the simple `if`. Its usage is as follows.

```
if predicate:
    code
```

The item `predicate` is any boolean-valued expression. If `predicate` evaluates to `True`, the code in the block executes. If not, the code in the block is skipped.

The `else` construct allows you to provide code that is executed if an `if` statement's predicate evaluates to `false`. Here is its usage.

```
if predicate:
    codeIfPredicateIsTrue
else:
    codeIfPredicateIsFalse
```

This puts a two-way fork in your code. One of the two alternatives *must* be executed.

The `elif` construct provides a means of producing a multi-way fork. Let us create an example. Imagine you are assigning grades. Let's do this. For A, B, C and D, you can have a + or - mark, but not for F. The input will be a percentage; if the last digit is 7,8 or 9, a + is granted. If it is a 0, 1, or 2, a - is given. Otherwise, there is no + or -. Let us build this.

So here is how we will get the score.

```
score = input("Enter a percentage:  ");
```

Run it and see that it works. We will write two functions, one for the letter grade, the other for the +/- modifier.

```
def letter_grade(n):
    if n < 60:
        out  = "F"
    elif n < 70:
        out  = "D"
    elif n < 80:
        out  = "C"
    elif n < 90:
        out  = "B"
    elif n <= 100:
        out  = "A"
    else:
        out = "Illegal grade!"
    return out
score = input("Enter a percentage:  ");
```

Run this and you are now reminded: cast things you get from `input` to a number type. For the sake of simplicity, we will use integers here.

```
unix> python grades.py
Enter a percentage:  95
Traceback (most recent call last):
  File "grades.py", line 16, in <module>
    print(letter_grade(score))
  File "grades.py", line 2, in letter_grade
    if n < 60:
TypeError: unorderable types: str() < int()
```

Just add this line

```
score = int(score)
```

The letter in the grade looks good.

```
unix> Feb 02:13:15:ppp> python grades.py
Enter a percentage:  95
A
unix> python grades.py
Enter a percentage:  90
A
Enter a percentage:  88
B
unix> python grades.py
Enter a percentage:  77
C
unix> python grades.py
Enter a percentage:  66
D
unix> python grades.py
Enter a percentage:  44
F
```

One lurking issue to watch for is to ensure a 100 gets an A+. We will make a second function to produce the modifier. Notice that if the student fails, he gets no modifier.

```
def modifier(n):
    out = ""
    if n >= 60:
        if n % 10 <= 2:
            out = "-"
```

```
        if n % 10 >= 8:
            out = "+"
    if n == 100:
        out = "+"
    return out
```

Here is some testing. You should check all branches and see to it that you are happy.

```
Enter a percentage:  100
A+
unix> python grades.py
Enter a percentage:  98
A+
unix> python grades.py
Enter a percentage:  94
A
unix> python grades.py
Enter a percentage:  91
A-
unix> python grades.py
Enter a percentage:  88
B+
unix> python grades.py
Enter a percentage:  59
F
unix> python grades.py
Enter a percentage:  61
D-
```

But what if the fool enters gibberish? Take exception. The fool! Spectate, videte, et ecce!

```
unix> python grades.py
Enter a percentage:  cats
Traceback (most recent call last):
  File "grades.py", line 28, in <module>
    score = int(score)
ValueError: invalid literal for int() with base 10: 'cats'
```

We will "try" to convert the input and punt if the `ValueError` rears its ugly head. Raising this error brings the program to a screeching halt at the scene of the crime. Here is the full code listing.

```
def letter_grade(n):
    if n < 60:
```

```
        out  = "F"
    elif n < 70:
        out  = "D"
    elif n < 80:
        out  = "C"
    elif n < 90:
        out  = "B"
    elif n <= 100:
        out  = "A"
    else:
        out = "Illegal grade!"
    return out
def modifier(n):
    out = ""
    if n >= 60:
        if n % 10 <= 2:
            out = "-"
        if n % 10 >= 8:
            out = "+"
    if n == 100:
        out = "+"
    return out
def grade(n):
    return letter_grade(n) + modifier(n)
score = input("Enter a percentage:  ")
try:
    score = int(score)
except ValueError:
    print("Illegal entry.  Try an integer 0-100.")
    quit()
print(grade(score))
```

You now begin to see how functions can call each other and how they can work as a team to separate a problem into manageable pieces that you can easily code. The function `grade` is an orchestrator that spits out the grade and which is really the only function the end-user ever calls.

So, now by example we see three conditional situations. The simple `if` just skips its code if its predicate is true. An `if-else` progression executes the `if` block if the predicate is true and the `else` block if the predicate is not true.

An `if-elif-else` progression keeps trying until a predicate is true. When it occurs, it executes that predicate's block and drops out of the progression. Is is good practice when using these to have an `else` block to handle any potential errors. Note that without an `else` block, the progression can go by and do nothing.

Programming Exercises Time for a date! Here are some calendar-oriented programming challenges. Functions shown are stubbed in so you can put this Python code in a file and it will run.

1. Write a function called `is_leap(year)` which returns `True` if the year leaps and which returns `False` otherwise. Here is the rule.
   - If a year is divisilble by 4 it leaps.
   - BUT every 100 years there is an exception.
   - BUT every 400 years there is an exception to the exception!
2. Here are some other functions to implement.

```
def date_plus(the_date):
    """Precondition: the_date is a string containing a date.
Postcondition: Return the sum of the year, month, and day.  You must accept any of the
dd/mm/yyyy
dd-mm-yyyy
ddmmyyyy
ddmmyy
dd/mm/yy
dd-mm-yy
"""
    pass
#here are some tests to implement.
date_plus("01/01/1970") == 1972
date_plus("08/12/1995") == 2015
date_plus("08/12/95") == 2015
date_plus("08/12/14") == 2034
```

3. ```
   def dayInYear(year, month, day):
       """prec:  year/month/day is a valid date
       postc: returns the ordinal position of the day in the year
       (Feb 15 is the 44th day of year 2000).
       Hint:  The list method sum is your friend. Learn about it."""
       return 0
   ```

4. ```
   def daysLeftInYear(year, month, day):
       """prec:  year/month/day is a valid date
       postc: returns the number of days left in the year
       (Feb 15 is the 44th day of year 2000)."""
       return 0
   ```

4 The Call Stack

When you run Python, it reads and stored all of your functions into memory. Basically functions are just assignments to variables in the global scope. In the

beginning `letter_grade`, `modifier`, and `grade` are read into memory. We then get to this code. Here we are in the "main routine" of our program.

```
score = input("Enter a percentage:  ")
try:
    score = int(score)
except ValueError:
    print("Illegal entry.  Try an integer 0-100.")
    quit()
print(grade(score))
```

There are two major pools of memory you must be concerned with when using Python. There is the heap, which is the warehouse storing all of your objects. Then there is the stack. The stack stores data structures called *activation records* or *stack frames*. A stack frame contains several pieces of information.

1. It stores a bookmark for exactly where it is in its execution. The bookmark advances as it executes.
2. It stores the parameters and the memory addresses of the objects passed to them.
3. It holds a local symbol table with all of the variables created inside of the function. The data from the parameters is loaded into this symbol table as well. When the function returns, this symbol table is exposed to be overwritten and it becomes inaccessible.
4. It stores a return address, and when it returns it goes to that address; at that time the function that called it (the caller) resumes its business.

When a function is executing, its local symbol table and the global symbol table (which contains the other functions) are visible.

The first line makes a function call to `input`. This call is placed on the stack. The `input` function waits for the user to type in an entry; when the enter key is hit, it returns the string the user typed in. Once the function returns, its stack frame is exposed to be overwritten; for all practical purposes it is destroyed. It disappears with no trace.

Then we perform the cast, which is really a function call. If the user types in gibberish, a `ValueError` is raised and, if there is no `except` block, the program dies right on the spot, right in the middle of the call to the `input` function.

Now let us suppose that we successfully get an integer from the user. A call is made to `print`. Right in the middle of `print` trying to do its job, we call `grade` and pass it `score`. So, on top of `print`'s stack frame we plop a stack frame for `grade`.

This frame stores a return address so it can get back to where we left off in

`print`'s execution. Now `grade` is running and it can see `n` in its private symbol table, which contains the memory address where `score` is living on the heap.

The first line of `grade` has a call to `letter_grade` in it; this call is being passed `n` so it gets a copy of the memory address where `score` is stored. You see a local variable `out` that gets created. The appropriate branch hands `out` a letter grade. The actual letter is created on the heap; `out` merely knows its memory address. That gets sent back to its caller, `grade`. This process is repeated for `modifier`.

With both parts of its return value in place, `grade` returns a grade with its modifier to `print`, which ships the result to `stdout` and returns. The program then ends. Throughout this book we will adhere to this style convention: *All functions should be defined before any other code is placed in a program.* This keeps things sane.

So to summarize, as a program runs, the following happen.

1. Python reads the program from top to bottom.
2. As functions are defined, they are placed in heap memory and the memory address where they are stored goes into the global symbol table.
3. The first unindented line after the functions are seen is the beginning of the program's "main routine." Variables created in the main routine go in the global symbol table. Since the main routine is last, variables in it cannot be seen by the functions without a little extra code. This is bad practice; we will refrain from doing this.
4. When a function is called, its stack frame goes on the call stack. The function puts the function's parameters and local variables in a local symbol table, which resides in its stack frame. The stack frame also maintains a return address to go back to once it returns, and a bookmark of where it is in its progress. If a function calls another function, a stack frame is created for that function and it is placed on the stack.
5. When a function returns, its stack frame is popped (removed) from the call stack and it is no longer accessible. All of the function's local variables die.
6. The stack will grow and shrink as the program runs.
7. When the main routine returns, program execution is over. The OS reclaims the space occupied by your program.

Recommendation A smart way to make programs is to place your main routine inside of a function called `main` and just call `main` in the main routine. Here is an example. In this way, the *only* occupants of the global symbol table are functions. The only worker statement in the main routine is `main()`. Here is a modest example.

```
def greet(name):
    return "Hello, " + name
def main()
    print(greet("General Grant"))
main()
```

The only items in your global symbol table are functions.

5 Python Standard Library Modules

A *module* is a Python program. Modules in the standard library consist largely of functions and constants. There are a plethora of these that accomplish a wide array of very useful tasks. We will learn here how to use a standard library module and how to read its documentation. We shall begin with the library `math`, and then take a peek into the very useful `os` module. Open a Python session. Use the `import` statement to make a library visible.

```
>>> import math
```

To see its contents, use the built-in function `dir`.

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

You can see some familiar looking things. Here are some constants. Note that they are prefixed by `math.`.

```
>>> math.e
2.718281828459045
>>> math.pi
3.141592653589793
```

Here we test-drive the logarithmic functions

```
>>> math.log2(1024)
10.0
>>> math.log10(1000)
```

```
3.0
>>> math.log(math.e)
1.0
```

Now let us see help on these functions.

```
>>> print(math.log.__doc__)
log(x[, base])

Return the logarithm of x to the given base.
If the base not specified, returns the natural logarithm (base e) of x.
>>> print(math.log2.__doc__)
log2(x)

Return the base 2 logarithm of x.
>>> print(math.log10.__doc__)
log10(x)

Return the base 10 logarithm of x.
```

We see a mysterious function log1p. Let us plumb the depths.

```
>>> print(math.log1p.__doc__)
log1p(x)

Return the natural logarithm of 1+x (base e).
The result is computed in a way which is accurate for x near zero.
```

The thoughtful makers of the standard libraries provided docstrings for their functions. When in doubt about a function, this is a quick handy reference. Don't be shy.

Programming Exercises

1. Write a function `sind` that computes sines using degree angle measure. Do the same for `cos` and `tan`. Make the best available use of standard library functions.
2. What do `floor` and `ceil` do?
3. For a how large an argument can you compute `math.factorial`?

5.1 Accessing your File System

Let us learn about another standard library, `os`. This is very handy for interacting with your file system. Begin by importing it and viewing its contents.

```
>>> import os
>>> dir(os)
['CLD_CONTINUED', 'CLD_DUMPED', 'CLD_EXITED', 'CLD_TRAPPED',
'EX_CANTCREAT', 'EX_CONFIG', 'EX_DATAERR', 'EX_IOERR', 'EX_NOHOST',
'EX_NOINPUT', 'EX_NOPERM', 'EX_NOUSER', 'EX_OK', 'EX_OSERR', 'EX_OSFILE',
'EX_PROTOCOL', 'EX_SOFTWARE', 'EX_TEMPFAIL', 'EX_UNAVAILABLE', 'EX_USAGE',
'F_LOCK', 'F_OK', 'F_TEST', 'F_TLOCK', 'F_ULOCK', 'MutableMapping',
'NGROUPS_MAX', 'O_ACCMODE', 'O_APPEND', 'O_ASYNC', 'O_CLOEXEC', 'O_CREAT',
'O_DIRECTORY', 'O_DSYNC', 'O_EXCL', 'O_EXLOCK', 'O_NDELAY', 'O_NOCTTY',
'O_NOFOLLOW', 'O_NONBLOCK', 'O_RDONLY', 'O_RDWR', 'O_SHLOCK', 'O_SYNC',
'O_TRUNC', 'O_WRONLY', 'PRIO_PGRP', 'PRIO_PROCESS', 'PRIO_USER', 'P_ALL',
'P_NOWAIT', 'P_NOWAITO', 'P_PGID', 'P_PID', 'P_WAIT', 'RTLD_GLOBAL',
'RTLD_LAZY', 'RTLD_LOCAL', 'RTLD_NODELETE', 'RTLD_NOLOAD', 'RTLD_NOW',
'R_OK', 'SCHED_FIFO', 'SCHED_OTHER', 'SCHED_RR', 'SEEK_CUR', 'SEEK_END',
'SEEK_SET', 'ST_NOSUID', 'ST_RDONLY', 'TMP_MAX', 'WCONTINUED', 'WCOREDUMP',
'WEXITED', 'WEXITSTATUS', 'WIFCONTINUED', 'WIFEXITED', 'WIFSIGNALED',
'WIFSTOPPED', 'WNOHANG', 'WNOWAIT', 'WSTOPPED', 'WSTOPSIG', 'WTERMSIG',
'WUNTRACED', 'W_OK', 'X_OK', '_DummyDirEntry', '_Environ', '__all__',
'__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_dummy_scandir', '_execvpe',
'_exists', '_exit', '_get_exports_list', '_putenv', '_spawnvef',
'_unsetenv', '_wrap_close', 'abort', 'access', 'altsep', 'chdir',
'chflags', 'chmod', 'chown', 'chroot', 'close', 'closerange', 'confstr',
'confstr_names', 'cpu_count', 'ctermid', 'curdir', 'defpath',
'device_encoding', 'devnull', 'dup', 'dup2', 'environ', 'environb',
'errno', 'error', 'execl', 'execle', 'execlp', 'execlpe', 'execv',
'execve', 'execvp', 'execvpe', 'extsep', 'fchdir', 'fchmod', 'fchown',
'fdopen', 'fork', 'forkpty', 'fpathconf', 'fsdecode', 'fsencode', 'fstat',
'fstatvfs', 'fsync', 'ftruncate', 'get_blocking', 'get_exec_path',
'get_inheritable', 'get_terminal_size', 'getcwd', 'getcwdb', 'getegid',
'getenv', 'getenvb', 'geteuid', 'getgid', 'getgrouplist', 'getgroups',
'getloadavg', 'getlogin', 'getpgid', 'getpgrp', 'getpid', 'getppid',
'getpriority', 'getsid', 'getuid', 'initgroups', 'isatty', 'kill',
'killpg', 'lchflags', 'lchmod', 'lchown', 'linesep', 'link', 'listdir',
'lockf', 'lseek', 'lstat', 'major', 'makedev', 'makedirs', 'minor',
'mkdir', 'mkfifo', 'mknod', 'name', 'nice', 'open', 'openpty', 'pardir',
'path', 'pathconf', 'pathconf_names', 'pathsep', 'pipe', 'popen', 'pread',
'putenv', 'pwrite', 'read', 'readlink', 'readv', 'remove', 'removedirs',
'rename', 'renames', 'replace', 'rmdir', 'scandir',
'sched_get_priority_max', 'sched_get_priority_min', 'sched_yield',
'sendfile', 'sep', 'set_blocking', 'set_inheritable', 'setegid', 'seteuid',
'setgid', 'setgroups', 'setpgid', 'setpgrp', 'setpriority', 'setregid',
'setreuid', 'setsid', 'setuid', 'spawnl', 'spawnle', 'spawnlp', 'spawnlpe',
'spawnv', 'spawnve', 'spawnvp', 'spawnvpe', 'st', 'stat',
'stat_float_times', 'stat_result', 'statvfs', 'statvfs_result', 'strerror',
'supports_bytes_environ', 'supports_dir_fd', 'supports_effective_ids',
```

```
'supports_fd', 'supports_follow_symlinks', 'symlink', 'sync', 'sys',
'sysconf', 'sysconf_names', 'system', 'tcgetpgrp', 'tcsetpgrp',
'terminal_size', 'times', 'times_result', 'truncate', 'ttyname', 'umask',
'uname', 'uname_result', 'unlink', 'unsetenv', 'urandom', 'utime', 'wait',
'wait3', 'wait4', 'waitpid', 'walk', 'write', 'writev']
```

Wow! The abundance of things is overwhelming. Let us play with a few. Here we learn our Python process's current working directory and we list its contents.

```
>>> os.listdir()
['.pp1core.tex.swp', '_minted-p0', '_minted-p1', 'bmp', 'bol', 'bv',
'p0.aux', 'p0.log', 'p0.out', 'p0.pdf', 'p0.tex', 'p0.toc', 'p0Code',
'p1.aux', 'p1.log', 'p1.out', 'p1.pdf', 'p1.tex', 'p1.toc', 'p1Code',
'piCode', 'pp0core.tex', 'pp1core.tex', 'rectangle.py']
>>> os.getcwd()
'/Users/morrison/book/ppp'
```

Now here is something that is interesting.

```
>>> print(os.path.__doc__)
Common operations on Posix pathnames.

Instead of importing this module directly, import os and refer to
this module as os.path.  The "os.path" name is an alias for this
module on Posix systems; on other systems (e.g. Mac, Windows),
os.path provides the same operations in a manner specific to that
platform, and is an alias to another module (e.g. macpath, ntpath).

Some of this can actually be useful on non-Posix systems too, e.g.
for manipulation of the pathname component of URLs.
>>> dir(os.path)
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_get_sep', '_joinrealpath', '_varprog',
'_varprogb', 'abspath', 'altsep', 'basename', 'commonpath', 'commonprefix',
'curdir', 'defpath', 'devnull', 'dirname', 'exists', 'expanduser',
'expandvars', 'extsep', 'genericpath', 'getatime', 'getctime', 'getmtime',
'getsize', 'isabs', 'isdir', 'isfile', 'islink', 'ismount', 'join', 'lexists',
'normcase', 'normpath', 'os', 'pardir', 'pathsep', 'realpath', 'relpath',
'samefile', 'sameopenfile', 'samestat', 'sep', 'split', 'splitdrive',
'splitext', 'stat', 'supports_unicode_filenames', 'sys']
>>>
```

Look at the "is" functions. We can test and see if an item present is a directory with `isdir`. Notice we can peek inside, too, using `os.listdir`.

```
>>> os.path.exists("p1.pdf")
True
>>> os.path.isdir("p1Code")
True
>>> os.listdir("p1Code")
['.grades.py.swp', 'docstring.py', 'evilScope.py', 'grades.py']
```

This module gives you all sorts of great access to your file system. You should experiment with it and see what you can do. Be careful: you can delete files with it! Explore it and its submodule `os.path`.

Programming Exercises

1. How can you compute the size of a file?
2. Write a function called getPermissionString(file) that does the following
   - It prints out an error message if the file does not exist.
   - If the file exists, it prints its permission string (do an ls -l to remind yourself what this looks like)
3. How do you find the absolute path of a file or directory?

5.2 Some Random Thoughts

If you ever want to write a game, you will want the ability to deal with random phenomena, such as the drawing of a card from a deck or rolling dice.

The Python library `random` is a powerful tool for accomplishing this sort of task. This library has the capability of generating *pseudo-random* numbers. There is a whole industry behind the generation of random numbers. Computers acutally use determininstic formulae to generate random numbers, hence the term pseudo-random.

Let us have a peek at what is inside of this module with our old friend `dir`.

```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random',
    'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_BuiltinMethodType',
    '_MethodType', '_Sequence', '_Set', '__all__', '__builtins__',
    '__cached__', '__doc__', '__file__', '__loader__', '__name__',
    '__package__', '__spec__', '_acos', '_bisect', '_ceil', '_cos',
    '_e', '_exp', '_inst', '_itertools', '_log', '_pi', '_random',
    '_sha512', '_sin', '_sqrt', '_test', '_test_generator',
    '_urandom', '_warn', 'betavariate', 'choice', 'choices',
    'expovariate', 'gammavariate', 'gauss', 'getrandbits',
```

```
    'getstate', 'lognormvariate', 'normalvariate', 'paretovariate',
    'randint', 'random', 'randrange', 'sample', 'seed', 'setstate',
    'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
    'weibullvariate']
>>>
```

Suppose you want to roll a pair of dice. You will want to get a tuple whose entries are random numbers 1-6. The `randint` function is just the tool for the job. Here we see its docstring.

```
>>> print(random.randint.__doc__)
Return random integer in range [a, b], including both end points.

>>>
```

The roll of a die gives us a number 1-6, so we can roll a die with a call to `random.randint(1,6)`. Let's have a look by running it a few times.

```
>>> import random
>>> random.randint(1,6)
4
>>> random.randint(1,6)
3
>>> random.randint(1,6)
1
>>> random.randint(1,6)
1
>>> random.randint(1,6)
4
>>> random.randint(1,6)
```

We can now create a function to roll a pair of dice. Create the program `dice.py`

```
import random
def roll_dice():
    return (random.randint(1,6), random.randint(1,6))
for k in range(6):
    print(roll_dice())
```

Run and see this. Your result, of course, will likely be different.

```
unix> python dice.py
(5, 6)
```

```
(4, 6)
(5, 6)
(2, 4)
(4, 6)
(1, 3)
```

Another useful method is `random.choice`, which will pick a random element out of a sequence. Here we create a function that tosses a coin.

```
def toss():
    return random.choice(["H", "T"])
```

Put this in a file and call `toss` repatedly. Two related method are `shuffle` and `sample`. We demonstrate them here.

```
>>> team = ["Pete", "Jane", "Evelyn", "Mario", "Maria", "Edward"]
>>> import random
>>> random.shuffle(team)
>>> team
['Edward', 'Evelyn', 'Jane', 'Mario', 'Maria', 'Pete']
>>> random.shuffle(team)
>>> team
['Edward', 'Jane', 'Maria', 'Pete', 'Evelyn', 'Mario']
>>> random.sample(team, 3)
['Mario', 'Evelyn', 'Edward']
>>> random.sample(team, 3)
['Maria', 'Mario', 'Evelyn']
>>> random.sample(team, 3)
['Maria', 'Evelyn', 'Edward']
```

The `sample` method picks a sample without replacement.

Programming Exercises

6 Recursion

Functions can call themselves, and this can often be useful. In fact, we can achieve repetition with this mechanism. Here is an example.

```
def rectangle(width, height, ch):
    if height == 0:
        return
    print(ch* width)
```

```
    rectangle(width, height - 1, ch)
rectangle(5,6, "*")
```

Let us run this for the doubting Thomases out there.

```
unix> python rectangle.py
*****
*****
*****
*****
*****
*****
```

So, what happened? To understand this we must analyze the call stack. Begin by adding a line to our code to see the local symbols that are visible.

```
def rectangle(width, height, ch):
    if height == 0:
        return
    print(dir())
    print(ch* width)
    rectangle(width, height - 1, ch)
rectangle(5,6, "*")
```

Running this we see the following.

```
unix> Feb 03:11:32:p1Code> python rectangle.py
['ch', 'height', 'width']
*****
['ch', 'height', 'width']
*****
['ch', 'height', 'width']
*****
['ch', 'height', 'width']
*****
['ch', 'height', 'width']
*****
['ch', 'height', 'width']
*****
unix>
```

Now let us print out the values held by these symbols. Delete the `dir` line and do this.

```
def rectangle(width, height, ch):
    if height == 0:
```

```
        return
    print("ch = %s, height = %s, width = %s" % (ch, height, width))
    print(ch* width)
    rectangle(width, height - 1, ch)
rectangle(5,6, "*")
```

Now run this to see more bread crumbs.

```
unix> python rectangle.py
ch = *, height = 6, width = 5
*****
ch = *, height = 5, width = 5
*****
ch = *, height = 4, width = 5
*****
ch = *, height = 3, width = 5
*****
ch = *, height = 2, width = 5
*****
ch = *, height = 1, width = 5
*****
unix>
```

We can now begin to see what has happened. The call to `rectangle(5, 6, "*")` causes a line of stars to print, then it spawns a call to `rectangle(4, 6, "*")`. The call to `rectangle(4, 6, "*")` causes a line of stars to print, then it spawns a call to `rectangle(3, 6, "*")`. This continues until we have a call to `rectangle(1,6,"*")` which spawns a call to `rectangle(0, 6, "*")`

We have now arrived at what is called the *base case* of our recursive function. Since the height is now 0, this function immediately returns. Where are we now? We are at the end of the call `rectangle(1, 6, "*")`. Since we already printed the stars and we made the function call inside, there is no more code. As a result, the function does a tacit return. This function now returns to the call `rectangle(2, 6, "*")`. This process continues until the original call returns and we are back in the main routine. The one function call in the main routine was done, so the main routine returns and our program's execution is over. What is left over? Rows of stars.

How about printing a list? Here is a tip: to print a list, print do nothing if the list is empty; otherwise print the first element, and then the rest of the elements. If you think this way, the recursive recipe is simple.

```
def printList(x):
    if x == []:
        return
```

```
    first = x[0]
    rest = x[1:]
    print(first)
    printList(rest)
def main():
    test = [1, 2, 3, 4, 5, 6]
    printList(test)
main()
```

Programming Exercises

1. Move one *entire* line of code so the list prints in the reverse order. Can you also find another way to print the list backwards?
2. Use `os.listdir` to get the contents of your `cwd`. Use `printList` to print it out.
3. Can you make it print in asciicographical order? reverse asciicographical order?
4. Can you write a recursive function that computes the sum of a list of numbers? the concatenation of a list of strings?
5. Can you write a function that accepts a list of numbers and returns its product?

7 Termnology Roundup

- **base case** This is a case in a recursive function that causes its ultimate return.
- **block** This is one or more lines of code that is indented a tabstop.
- **boss statement** This is a statement that controls the flow of a program. It must own a block of code, and it is a grammatically incomplete sentence.
- **call** To use a function.
- **docstring** This is a string in the first line of a function that gives information on the function's action.
- **function scope** Varibles created inside of functions and parameters of functions are invisible when the function is not executing.
- **global scope** Variables with global scope are visible at all times.
- **module** This is a file with Python code.
- **parameter** This is a value that is passed to a function.
- **pass** This is the act of sending an argument of a function to the function's code when the function is called.

- **postcondition** This is what is true when a function is done executing.
- **precondition** A condition that should inhere before we call a function
- **return value** The object, if any, that is returned by the function?
- **side-effect** This refers to actions whose consequences persist beyond the lifetime of a function.
- **worker statement** This is a Python statement containing executable code. Grammatically, it is a complete sentence.