installed, you need to find Python in the appropriate menu. Windows users may choose to run Python in a command shell (i.e., a DOS window) where it will behave very similarly to Linux or OS X.

For all three operating systems (Linux, OS X, Windows) there is also an integrated development environment for Python named `IDLE`. If interested, you may download and install this on your computer.[2] For help on getting started with IDLE see `http://hkn.eecs.berkeley.edu/~dyoo/python/idle_int`

Once Python starts running in interpreter mode, using `IDLE` or a command shell, it produces a prompt, which waits for your input. For example, this is what I get when I start Python in a command shell on my Linux box:

```
doty@brauer:~% python
Python 2.5.2 (r252:60911, Apr 21 2008, 11:12:42)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

where the three symbols `>>>` indicates the prompt awaiting my input.

So experiment, using the Python interpreter as a calculator. Be assured that you cannot harm anything, so play with Python as much as you like. For example:

```
>>> 2*1024
2048
>>> 3+4+9
16
>>> 2**100
1267650600228229401496703205376L
```

In the above, we first asked for the product of 2 and 1024, then we asked for the sum of 3, 4, and 9 and finally we asked for the value of 2^{100}. Note that multiplication in Python is represented by $*$, addition by $+$, and exponents by `**`; you will need to remember this syntax. The L appended to the last answer is there to indicate that this is a *long* integer; more on this later. It is also worth noting that Python does arbitrary precision integer arithmetic, by default:

```
>>> 2**1000
10715086071862673209484250490600018105614048117055336074437503
88837035105112493612249319837881569585812759467291755314682518
87145285692314043598457757469857480393456777482423098542107460
05062371141877954182153046474983581941267398767559165543946077
06629145711964776865421676604298316526243868372056680693769L
```

Here is another example, where we print a table of perfect squares:

```
>>> for n in [1,2,3,4,5,6]:
...   print n**2
...
1
4
9
16
25
36
```

[2]Both Python and IDLE should be already preinstalled on all Loyola Windows computers.

This illustrates several points. First, the expression [1,2,3,4,5,6] is a list, and we print the values of n^2 for n varying over the list. If we prefer, we can print horizontally instead of vertically:

```
>>> for n in [1,2,3,4,5,6]:
...   print n**2,
...
1 4 9 16 25 36
```

simply by adding a comma at the end of the print command, which tells Python *not* to move to a new line before the next print.

These last two examples are examples of a *compound* command, where the command is divided over two lines (or more). That is why you see `...` on the second line instead of the usual `>>>`, which is the interpreter's way of telling us it awaits the rest of the command. On the third line we entered nothing, in order to tell the interpreter that the command was complete at the second line. Also notice the *colon* at the end of the first line, and the *indentation* in the second line. Both are required in compound Python commands.

2.2 Quitting the interpreter

In a terminal you can quit a Python session by CTRL-D. (Hold down the CTRL key while pressing the D key.) In IDLE you can also quit from the menu.

If the interpreter gets stuck in an infinite loop, you can quit the current execution by CTRL-C.

2.3 Loading commands from the library

Python has a very extensive library of commands, documented in the *Python Library Reference Manual* [2]. These commands are organized into *modules*. One of the available modules is especially useful for us: the `math` module. Let's see how it may be used.

```
>>> from math import sqrt, exp
>>> exp(-1)
0.36787944117144233
>>> sqrt(2)
1.4142135623730951
```

We first `import` the `sqrt` and `exp` functions from the `math` module, then use them to compute $e^{-1} = 1/e$ and $\sqrt{2}$.

Once we have loaded a function from a module, it is available for the rest of that session. When we start a new session, we have to reload the function if we need it.

Note that we could have loaded both functions `sqrt` and `exp` by using a wildcard `*`:

```
>>> from math import *
```

which tells Python to import *all* the functions in the `math` module.

What would have happened if we forgot to import a needed function? After starting a new session, if we type

```
>>> sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

we see an example of an error message, telling us that Python does not recognize `sqrt`.

2.4 Defining functions

It is possible, and very useful, to define our own functions in Python. Generally speaking, if you need to do a calculation only once, then use the interpreter. But when you or others have need to perform a certain type of calculation many times, then define a function. For a simple example, the compound command

```
>>> def f(x):
...     return x*x
...
```

defines the squaring function $f(x) = x^2$, a popular example used in elementary math courses. In the definition, the first line is the function header where the name, `f`, of the function is specified. Subsequent lines give the body of the function, where the output value is calculated. Note that the final step is to `return` the answer; without it we would never see any results. Continuing the example, we can *use* the function to calculate the square of any given input:

```
>>> f(2)
4
>>> f(2.5)
6.25
```

The name of a function is purely arbitrary. We could have defined the same function as above, but with the name `square` instead of `f`; then to use it we use the new function name instead of the old:

```
>>> def square(x):
...     return x*x
...
>>> square(3)
9
>>> square(2.5)
6.25
```

Actually, a function name is not completely arbitrary, since we are not allowed to use a *reserved word* as a function name. Python's reserved words are: `and`, `def`, `del`, `for`, `is`, `raise`, `assert`, `elif`, `from`, `lambda`, `return`, `break`, `else`, `global`, `not`, `try`, `class`, `except`, `if`, `or`, `while`, `continue`, `exec`, `import`, `pass`, `yield`.

By the way, Python also allows us to define functions using a format similar to the *Lambda Calculus* in mathematical logic. For instance, the above function could alternatively be defined in the following way:

```
>>> square = lambda x: x*x
```

Here `lambda x: x*x` is known as a lambda expression. Lambda expressions are useful when you need to define a function in just one line; they are also useful in situations where you need a function but don't want to name it.

Usually function definitions will be stored in a module (file) for later use. These are indistinguishable from Python's Library modules from the user's perspective.

2.5 Files

Python allows us to store our code in files (also called modules). This is very useful for more serious programming, where we do not want to retype a long function definition from the very beginning just to change one mistake. In doing this, we are essentially defining our own modules, just like the modules defined already in the Python library. For example, to store our squaring function example in a file, we can use any text editor[3] to type the code into a file, such as

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

Notice that we omit the prompt symbols `>>>`, `...` when typing the code into a file, but the indentation is still important. Let's save this file under the name "SquaringFunction.py" and then open a terminal in order to run it:

```
doty@brauer:~% python
Python 2.5.2 (r252:60911, Apr 21 2008, 11:12:42)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>> from SquaringFunction import square
>>> square(1.5)
2.25
```

Notice that I had to import the function from the file before I could use it. Importing a command from a file works exactly the same as for library modules. (In fact, some people refer to Python files as "modules" because of this analogy.) Also notice that the file's extension (`.py`) is omitted in the `import` command.

2.6 Testing code

As indicated above, code is usually developed in a file using an editor. To test the code, import it into a Python session and try to run it. Usually there is an error, so you go back to the file, make a correction, and test again. This process is repeated until you are satisfied that the code works. The entire process is known as the *development cycle*.

There are two types of errors that you will encounter. *Syntax* errors occur when the form of some command is invalid. This happens when you make typing errors such as misspellings, or call something by the wrong name, and for many other reasons. Python will always give an error message for a syntax error.

2.7 Scripts

If you use Mac OS X or some other variant of Unix (such as Linux) then you may be interested in running Python commands as a script. Here's an example. Use an editor to create a file name SayHi containing the following lines

```
#! /usr/bin/python
print "Hello World!"
print "- From your friendly Python program"
```

[3]Most developers rely on `emacs` for editing code. Other possible choices are `Notepad` for Windows, `gedit` for Linux/Gnome, and `TextEdit` for OS X. `IDLE` comes with its own editor, by the way.

The first line tells Python that this is a script. After saving the file, make it executable by typing `chmod 755 SayHi` in the terminal. To run the script, type `./SayHi` in the terminal. Note that if you move the script someplace in your search path, then you can run it simply by typing `SayHi`. Type `echo $PATH` to see what folders are in your search path, and type `which python` to see where your python program is — this should match the first line in your script.

As far as I know, it is impossible to run Python scripts in a similar way on a Windows machine.

3 Python commands

3.1 Comments

In a Python command, anything after a `#` symbol is a comment. For example:

```
print "Hello world" #this is silly
```

Comments are not part of the command, but rather intended as documentation for anyone reading the code.

Multiline comments are also possible, and are enclosed by triple double-quote symbols:

```
"""This is an example of a long comment
that goes on
and on
and on."""
```

3.2 Numbers and other data types

Python recognizes several different types of data. For instance, 23 and -75 are *integers*, while 5.0 and -23.09 are *floats* or *floating point numbers*. The type float is (roughly) the same as a real number in mathematics. The number 12345678901 is a *long integer*; Python prints it with an "L" appended to the end.

Usually the type of a piece of data is determined implicitly.

3.2.1 The `type` function

To see the type of some data, use Python's builtin `type` function:

```
>>> type(-75)
<type 'int'>
>>> type(5.0)
<type 'float'>
>>> type(12345678901)
<type 'long'>
```

Another useful data type is *complex*, used for complex numbers. For example:

```
>>> 2j
2j
>>> 2j-1
(-1+2j)
>>> complex(2,3)
```

```
(2+3j)
>>> type(-1+2j)
<type 'complex'>
```

Notice that Python uses `j` for the complex unit (such that $j^2 = -1$) just as physicists do, instead of the letter i preferred by mathematicians.

3.2.2 Strings

Other useful data types are *strings* (short for "character strings"); for example `"Hello World!"`. Strings are sequences of characters enclosed in single or double quotes:

```
>>> "This is a string"
'This is a string'
>>> 'This is a string, too'
'This is a string, too'
>>> type("This is a string")
<type 'str'>
```

Strings are an example of a *sequence* type.

3.2.3 Lists and tuples

Other important sequence types used in Python include *lists* and *tuples*. A sequence type is formed by putting together some other types in a sequence. Here is how we form lists and tuples:

```
>>> [1,3,4,1,6]
[1, 3, 4, 1, 6]
>>> type( [1,3,4,1,6] )
<type 'list'>
>>> (1,3,2)
(1, 3, 2)
>>> type( (1,3,2) )
<type 'tuple'>
```

Notice that lists are enclosed in square brackets while tuples are enclosed in parentheses. Also note that lists and tuples do not need to be homogeneous; that is, the components can be of different types:

```
>>> [1,2,"Hello",(1,2)]
[1, 2, 'Hello', (1, 2)]
```

Here we created a list containing four components: two integers, a string, and a tuple. Note that components of lists may be other lists, and so on:

```
>>> [1, 2, [1,2], [1,[1,2]], 5]
[1, 2, [1, 2], [1, [1, 2]], 5]
```

By nesting lists within lists in this way, we can build up complicated stuctures.

Sequence types such as lists, tuples, and strings are always *ordered*, as opposed to a set in mathematics, which is always unordered. Also, repetition is allowed in a sequence, but not in a set.

3.2.4 The `range` function

The `range` function is often used to create lists of integers. It has three forms. In the simplest form, `range(n)` produces a list of all numbers $0, 1, 2, \ldots, n-1$ starting with 0 and ending with $n-1$. For instance,

```
>>> range(17)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

You can also specify an optional starting point and an increment, which may be negative. For instance, we have

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

Note the use of a negative increment in the last example.

3.2.5 Boolean values

Finally, we should mention the *boolean* type. This is a value which is either `True` or `False`.

```
>>> True
True
>>> type(True)
<type 'bool'>
>>> False
False
>>> type(False)
<type 'bool'>
```

Boolean types are used in making decisions.

3.3 Expressions

Python expressions are not commands, but rather form part of a command. An expression is anything which produces a value. Examples of expressions are: `2+2`, `2**100`, `f((x-1)/(x+1))`. Note that in order for Python to make sense of the last one, the variable `x` must have a value assigned and `f` should be a previously defined function.

Expressions are formed from variables, constants, function evaluations, and operators. Parentheses are used to indicate order of operations and grouping, as usual.

3.4 Operators

The common binary operators for arithmetic are `+` for addition, `-` for subtraction, `*` for multiplication, and `/` for division. As already mentioned, Python uses `**` for exponentiation. Integer division is performed so that the result is *always* another integer (the integer quotient):

```
>>> 25/3
8
>>> 5/2
2
```

This is a wrinkle that you will always have to keep in mind when working with Python. To get a more accurate answer, use the float type:

```
>>> 25.0/3
8.3333333333333339
>>> 5/2.0
2.5
```

If just one of the operands is of type float, then the result will be of type float. Here is another example of this pitfall:

```
>>> 2**(1/2)
1
```

where we wanted to compute the square root of 2 as the $\frac{1}{2}$ power of 2, but the division in the exponent produced a result of 0 because of integer division. A correct way to do this computation is:

```
>>> 2**0.5
1.4142135623730951
```

Another useful operator is `%`, which is read as "mod". This gives the remainder of an integer division, as in

```
>>> 5 % 2
1
>>> 25 % 3
1
```

which shows that 5 mod 2 = 1, and 25 mod 3 = 1. This operator is useful in number theory and cryptography.

Besides the arithmetic operators we need comparison operators: `<`, `>`, `<=`, `>=`, `==`, `!=`, `<>`. In order these are read as: *is less than*, *is greater than*, *is less than or equal to*, *is greater than or equal to*, *is equal to*, *is not equal to*, *is not equal to*. The result of a comparison is always a boolen value `True` or `False`.

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

Note that `!=` and `<>` are synonomous; either one means *not equal to*. Also, the operator `==` means *is equal to*.

```
>>> 2 <> 3
True
>>> 2 != 3
True
>>> 0 != 0
False
```

```
>>> 0 == 0
True
```

3.5 Variables and assignment

An assignment statement in Python has the form *variable* = *expression*. This has the following effect. First the expression on the right hand side is evaluated, then the result is assigned to the variable. After the assignment, the variable becomes a name for the result. The variable retains the same value until another value is assigned, in which case the previous value is lost. Executing the assignment produces no output; its purpose it to make the association between the variable and its value.

```
>>> x = 2+2
>>> print x
4
```

In the example above, the assignment statement sets x to 4, producing no output. If we want to see the value of x, we must print it. If we execute another assignment to x, then the previous value is lost.

```
>>> x = 380.5
>>> print x
380.5
>>> y = 2*x
>>> print y
761.0
```

Remember: A single `=` is used for assignment, the double `==` is used to test for equality.

In mathematics the equation $x = x + 1$ is nonsense; it has no solution. In computer science, the statement `x = x + 1` is useful. Its purpose is to add 1 to x, and reassign the result to x. In short, x is *incremented* by 1.

```
>>> x = 10
>>> x = x + 1
>>> print x
11
>>> x = x + 1
>>> print x
12
```

Variable names may be any contiguous sequence of letters, numbers, and the underscore (_) character. The first character must not be a number, and you may not use a reserved word as a variable name. Case is important; for instance `Sum` is a different name than `sum`. Other examples of legal variable names are: `a`, `v1`, `v_1`, `abc`, `Bucket`, `monthly_total`, `__pi__`, `TotalAssets`.

3.6 Decisions

The `if`–`else` is used to make choices in Python code. This is a compound statement. The simplest form is

```
if condition:
  action-1
```

```
else:
  action-2
```

The indentation is required. Note that the else and its action are optional. The actions *action-1* and *action-2* may consist of many statements; they must all be indented the same amount. The *condition* is an expression which evaluates to `True` or `False`.

Of course, if the condition evaluates to `True` then *action-1* is executed, otherwise *action-2* is executed. In either case execution continues with the statement after the if-else. For example, the code

```
x = 1
if x > 0:
  print "Friday is wonderful"
else:
  print "Monday sucks"
print "Have a good weekend"
```

results in the output

```
Friday is wonderful
Have a good weekend
```

Note that the last print statement is not part of the if-else statement (because it isn't indented), so if we change the first line to say `x = 0` then the output would be

```
Monday sucks
Have a good weekend
```

More complex decisions may have several alternatives depending on several conditions. For these the `elif` is used. It means "else if" and one can have any number of `elif` clauses between the `if` and the `else`. The usage of `elif` is best illustrated by an example:

```
if x >= 0 and x < 10:
  digits = 1
elif x >= 10 and x < 100:
  digits = 2
elif x >= 100 and x < 1000:
  digits = 3
elif x >= 1000 and x < 10000:
  digits = 4
else:
  digits = 0  # more than 4
```

In the above, the number of digits in `x` is computed, so long as the number is 4 or less. If `x` is negative or greater than 10000, then `digits` will be set to zero.

3.7 Loops

Python provides two looping commands: `for` and `while`. These are compound commands.

3.7.1 `for` loop

The syntax of a `for` loop is

```
for item in list:
  action
```

As usual, the *action* consists of one or more statements, all at the same indentation level. These statements are also known as the *body* of the loop. The *item* is a variable name, and *list* is a list.

Execution of the `for` loop works by setting the variable successively to each item in the list, and then executing the body each time. Here is a simple example (the comma at the end of the print makes all printing occur on the same line):

```
for i in [2, 4, 6, 0]:
  print i,
```

This produces the output

```
2 4 6 0
```

3.7.2 `while` loop

The syntax of the `while` loop is

```
while condition:
  action
```

Of course, the *action* may consist of one or more statements all at the same indentation level. The statements in the *action* are known as the *body* of the loop. Execution of the loop works as follows. First the condition is evaluated. If `True`, the body is executed and the condition evaluated again, and this repeats until the condition evaluates to `False`. Here is a simple example:

```
n = 0
while n < 10:
  print n,
  n = n + 3
```

This produces the following output

```
0 3 6 9
```

Note that the body of a while loop is never executed if the condition evaluates to `False` the first time. Also, if the body does not change the subsequent evaluations of the condition, an infinite loop may occur. For example

```
while True:
  print "Hello",
```

will print Hellos endlessly. To interrupt the execution of an infinite loop, use CTRL-C.

3.7.3 `else` in loops

A loop may have an optional `else` which is executed when the loop finishes. For example, the loop

```
for n in [10,9,8,7,6,5,4,3,2,1]:
  print n,
else:
  print "blastoff"
```

results in the output

```
10 9 8 7 6 5 4 3 2 1 blastoff
```

and the loop

```
n=10
while n > 0:
  print n,
  n = n - 1
else:
  print "blastoff"
```

has the same effect (it produces identical output).

3.7.4 `break`, `continue`, and `pass`

The `break` statement, like in C, breaks out of the smallest enclosing for or while loop. The `continue` statement, also borrowed from C, continues with the next iteration of the loop. The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

Here is an example of the use of a `break` statement and an `else` clause in a loop.

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print n, 'equals', x, '*', n/x
            break
    else:
        # loop fell through without finding a factor
        print n, 'is a prime number'
```

The above code searches for prime numbers between 2 and 10, and produces the following output.

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

3.8 Lists

As already mentioned, a list is a finite sequence of items, and one could use the range function to create lists of integers.

In Python, lists are not required to be homogeneous, i.e., the items could be of different types. For example,

```
a = [2, "Jack", 45, "23 Wentworth Ave"]
```

is a perfectly valid list consisting of two integers and two strings. One can refer to the entire list using the identifier `a` or to the i-th item in the list using `a[i]`.

```
>>> a = [2, "Jack", 45, "23 Wentworth Ave"]
>>> a
[2, 'Jack', 45, '23 Wentworth Ave']
>>> a[0]
2
>>> a[1]
'Jack'
>>> a[2]
45
>>> a[3]
'23 Wentworth Ave'
```

Note that the numbering of list items *always begins at 0 in Python*. So the four items in the above list are indexed by the numbers 0, 1, 2, 3.

List items may be assigned a new value; this of course changes the list. For example, with `a` as above:

```
>>> a
[2, 'Jack', 45, '23 Wentworth Ave']
>>> a[0] = 2002
>>> a
[2002, 'Jack', 45, '23 Wentworth Ave']
```

Of course, the entire list may be assigned a new value, which does not have to be a list. When this happens, the previous value is lost:

```
>>> a
[2002, 'Jack', 45, '23 Wentworth Ave']
>>> a = 'gobbletygook'
>>> a
'gobbletygook'
```

3.8.1 Length of a list; empty list

Every list has a *length*, the number of items in the list, obtained using the `len` function:

```
>>> x = [9, 4, 900, -45]
>>> len(x)
4
```

Of special importance is the *empty list* of length 0. This is created as follows:

```
>>> x = []
>>> len(x)
0
```

3.8.2 Sublists (slicing)

Sublists are obtained by *slicing*, which works analogously to the `range` function discussed before. If `x` is an existing list, then `x[start:end]` is the sublist consisting of all items in the original list at index positions i such that

$$\texttt{start} \leq i < \texttt{end}.$$

Of course, we must remember that indexing items always starts at 0 in Python. For example,

```
>>> x=range(0,20,2)
>>> x
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> x[2:5]
[4, 6, 8]
>>> x[0:5]
[0, 2, 4, 6, 8]
```

When taking a slice, either parameter `start` or `end` may be omitted: if `start` is omitted then the slice consists of all items up to, but not including, the one at index position `end`, similarly, if `end` is omitted the slice consists of all items starting with the one at position `start`. For instance, with the list `x` as defined above we have

```
>>> x[:5]
[0, 2, 4, 6, 8]
>>> x[2:]
[4, 6, 8, 10, 12, 14, 16, 18]
```

In this case, `x[:5]` is equivalent to `x[0:5]` and `x[2:]` is equivalent to `x[2:len(x)]`.

There is an optional third parameter in a slice, which if present represents an increment, just as in the range function. For example,

```
>>> list = range(20)
>>> list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
>>> list[0:16:2]
[0, 2, 4, 6, 8, 10, 12, 14]
>>> list[0:15:2]
[0, 2, 4, 6, 8, 10, 12, 14]
```

Notice that one may cleverly use a *negative* increment to effectively reverse a list, as in:

```
>>> list[18::-1]
[17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

In general, the slice `x[len(x)::-1]` reverses any existing list `x`.

3.8.3 Joining two lists

Two existing lists may be *concatenated* together to make a longer list, using the `+` operator:

```
>>> [2,3,6,10] + [4,0,0,5,0]
[2, 3, 6, 10, 4, 0, 0, 5, 0]
```

3.8.4 List methods

If `x` is the name of an existing list, we can `append` an item `item` to the end of the list using

```
x.append(item)
```

For example,

```
>>> x = [3, 6, 8, 9]
>>> x.append(999)
>>> x
[3, 6, 8, 9, 999]
```

A similar method is called `insert`, which allows an element to be inserted in the list at a specified position:

```
>>> x = ['a', 'c', '3', 'd', '7']
>>> x.insert(0,100)
>>> x
[100, 'a', 'c', '3', 'd', '7']
>>> x.insert(3,'junk')
>>> x
[100, 'a', 'c', 'junk', '3', 'd', '7']
```

One can also delete the first occurrence of some item in the list (if possible) using `remove` as follows:

```
>>> x.remove('a')
>>> x
[100, 'c', 'junk', '3', 'd', '7']
```

To delete the item at index position i use `x.pop(i)`, as in:

```
>>> x.pop(0)
100
>>> x
['c', 'junk', '3', 'd', '7']
```

Notice that pop not only changes the list, but it also returns the item that was deleted. Also, by default `x.pop()` pops off the last item:

```
>>> x.pop()
'7'
>>> x
['c', 'junk', '3', 'd']
```

Many more methods exist for manipulating lists; consult the Python Tutorial [1] or Python Library Reference [2] for more details.

3.9 Strings

A string in Python is a sequence of characters. In some sense strings are similar to lists, however, there are important differences. One major difference is that Python strings are *immutable*, meaning that we are not allowed to change individual parts of them as we could for a list. So if `x` is an existing string, then `x[i]` gets the character at position i, but we are not allowed to reassign that character, as in `x[5] = 's'`.

```
>>> x = 'gobbletygook'
>>> x[2]
'b'
>>> x[5]
'e'
>>> x[5] = 's'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Just as for lists, string items are indexed starting at 0. Slicing for strings works exactly the same as for lists. The length function `len` is the same as for lists, and concatenation is the same too. But

the list methods `append`, `insert`, `delete`, and `pop` are *not available* for strings, because strings are immutable. If you need to change an existing string, you must make a new, changed, one.

There are many string methods for manipulating strings, documented in the Python Library Reference Manual [2]. For example, you can capitalize an existing string `x` using `x.capitalize()`; this returns a new copy of the string in which the first character has been capitalized.

```
>>> a = 'gobbletygook is refreshing'
>>> a.capitalize()
'Gobbletygook is refreshing'
```

Other useful methods are `find` and `index`, which are used to find the first occurence of a substring in a given string. See the manuals for details.

References

[1] Guido van Rossum, *Python Tutorial*, http://docs.python.org.

[2] Guido van Rossum, *Python Library Reference Manual*, http://docs.python.org.