Simple Data Types

There are a number of data types that are considered "primitive" in that they contain only a single value. These data types are int, long, float, complex, and bool.

int Integers (numbers without fractions). In Python 3 the int is the only integer data type. If an integer value is small enough that it fits into the hardware then the processor can operate on it directly at full speed; otherwise it will be operated on in software. Integers may be as large as desired, but with loss of performance.

long In Python 2, the hardware int type is distinct from software long integers. Long integers are handled in software, and may have any value (including those covered by the int data type), but at the cost of a loss of performance. In some versions of Python, the int is 32 bit, which maps onto the range from -2147483648 to +2147483647; any integer calculation which exceeds these limits automatically defaults over to the long data type. The switch-over point depends on the native size of integers on a processor (either 32 bits or 64 bits).

float Numbers with fractions. The legal range of values is approximately $10^{324}$ to $10^{308}$, with about 15 to 16 significant figures of precision. Calculations which exceed $10^{308}$ are converted to inf (infinity), and calculations which are smaller than $10^{-324}$ go to zero. It is usually not a good idea to compare a numeric value to a float because of round-off error (even when numbers are mathematically supposed to be equal, they rarely are in practice). Absorbs integers. For example, 5.0+3 is 8.0 (the 3 is cast as a float before the addition).

complex Numbers with a real part and an imaginary part (both floats). Contrary to math operations that use i to indicate the imaginary part, Python uses j instead. A number which is 5 along the real axis and 7 along the imaginary axis is written in Python as (5+7j). Adding an int or float to a complex adds the value to the real part. For example, (5+7j)+10 is (15+7j).

bool Short for Boolean (named after George Boole). Only has two possible values, False and True. The result of any comparison is always a bool. For example, 5<7 is True, but 5>7 is False. In a weird type violation, False is equivalent to 0 and True is equivalent to 1, so an expression such as 9+(5<7) is the same as 9+True, which equals 10 as the result.
Structured Data Types

There are a number of data types that are structures that can contain many values.

**list**
A list is the most general data structure in Python. Uses square brackets as the constructor. Is **mutable** (items in the list can be changed after the list has been constructed). Lists may contain items of any other type, including other lists. Can be defined explicitly, as in \( X = [4, \ 7.8, \ \text{"Frog"}, \ \text{True}] \), or by the range function. An empty list is denoted by `[]`.

**tuple**
Like a list in all ways except that it uses parentheses as the constructor and is **immutable**. Can be defined explicitly, as in \( X = (4, \ 7.8, \ \text{"Frog"}, \ \text{True}) \).

**string**
Also like a list in all ways except that it uses quotes (of various forms) as the constructor, contains only characters, and is **immutable**. Can be defined explicitly, as in \( X = \text{"Frog"} \) or by use of the \texttt{chr} function to convert integers in the range \( 0...255 \) to a character, as in \( X = \text{chr}(65) \) to assign the letter "A" to the variable \( X \). It is legal to use either 'single quotes' or "double quotes" as the constructor. May also use ''' triple-single ''' or """ triple-double """" quotes, and in these cases the string may span multiple lines and include the line breaks as part of the string.

In all three of these types components are indexed by integers in **square brackets**, starting at index 0 on the left and increasing to the right, or by \(-1\) on the right and decreasing to the left. For example, \( X[0] \) is the first item of the structure, while either \( X[-1] \) or \( X[\text{len}(X)-1] \) indicates the last item of the structure.

**Slicing** also applies to extract a subsection of the structure. For example, \( X[M:N] \) extracts the section of the structure starting at index \( M \) up through but not including index \( N \). Either \( M \) or \( N \) may be omitted, as in \( X[M:] \) (everything from index \( M \) to the end of the structure), \( X[:N] \) (everything from the beginning of the structure up through but not including index \( N \)), or \( X[:] \) (everything).

**dict**
A “hash table” of **key-value** pairs. Offers very fast look-up of a key. Keys may be any primitive type (int, float, bool, complex), strings, or tuples, but not lists. Values may be of any type, including other dictionaries. Constructor is curly braces. An empty dictionary is denoted by `{}`. To reference a stored value, use the dictionary name followed by the key in square brackets.

In all of the structured types the \texttt{len} function is used to tell how many items are present (items in a list or tuple, characters in a string, or keys in a dictionary).
Variables

Variable names can be of any length, but must start with a letter or underscore, and may contain letters or digits or the underscore. Upper case is distinct from lower case. It is almost never a good idea to have two variables with the same letters that differ only by case (frog and frog, for example).

Variables can appear on either side of the equal sign in an assignment statement, but expressions can only appear on the right side. For example, the expression $N = N + 1$ is legal, but the statement $N + 1 = N$ is illegal.

In a statement such as $N = N + 1$, the old value of $N$ is used in the calculation, and then the result becomes the new value of $N$. Typing the simple expression $N + 1$ on a line is also legal, and the proper value is computed, but that value is discarded since it is not assigned to a variable; the expression $N + 1$ by itself does not change $N$.

Variables are dynamically typed, which means that they take on the data type of whatever they are assigned. For example, the statement $N = 5$ makes $N$ an int, but if that statement is followed by $N = 4.5$ then $N$ becomes a float.
Basic Structures

In Python statements are normally written one per line, and program execution goes from one statement to the next in strict linear sequence. All statements in a block must have the same level of indentation. Statements that enclose other code such as the if and the while statement indent the statements that they enclose.

The if-elif-else statement

To make a selection, use the if statement (the if is required; the elif and else are not):

```python
if (condition):
    # do something here
elif (condition):
    # do something here
elif (condition):
    # do something here
else:
    # do something here
```

The while statement

To do something more than once, use a while loop:

```python
while (condition):
    # do something here
```

Note that while loops come in two “flavors”. One is open-ended in the number of times that the loop body executes, and the other is more fixed. There is no guarantee that the following loop will execute any particular number of times because it is dependent on user input:

```python
X = int(input("Enter a positive number"))
while (X < 0):
    X = int(input("Enter a positive number"))
```

but the following loop will execute exactly $N$ times (for any positive value of $N$):

```python
N = int(input("Enter a positive number"))
Counter = 0
while (Counter < N):
    # do something here for each value of Counter 0,1,2,...,N
    Counter = Counter + 1
```
The for statement

The last form of the while loop is best replaced by a for loop instead:

```python
N = int(input("Enter a positive number"))
for Counter in range(N):
    # do something here for each value of Counter 0,1,2,...,N
```

Note that in a for loop there is no explicit initialization or increment of Counter; that is handled by the for loop itself. The general form of a for loop is:

```python
for variable in structure:
    # do something here for each value of the variable
```

where structure can be a list (either an explicit list like [3,7,1,2] or a list generated by the range function), a tuple, a string (the variable will successively take on each character in the string), or a dictionary (the variable will successively take on each key in the dictionary).

Single-line bodies

If the body of an if, elif, else, while, or for statement consists of a single instruction, that instruction may appear on the same line as the controlling statement, as in:

```python
if (condition): # do something here
while (condition): # do something here
for variable in structure: # do something here
```

The pass statement

If an action is as of yet unknown, or it is intended that time elapses without anything happening, use the pass statement as a placeholder to keep Python happy:

```python
for variable in structure:
    pass
```
Functions and Parameter Passing

Functions have a meaning similar to that of mathematics. For example, the square root function in the math library `sqrt` receives a value through its argument list and returns a value through its name.

Basic ideas

If we were to write it ourselves we would set up a template such as:

```python
def MySqrt(N):
    # Compute the result
    # and put the value into
    # variable Result
    return Result
```

Note that the body of the function is indented relative to the `def` statement. This function has one parameter, called `N`, which is local to the function. That is, it is a socket into which a value is plugged from outside when the function is called, but the value of `N` vanishes when the function is complete. There can be other variables named `N` elsewhere, but they have nothing to do with this `N`. Calls to this function can be any of the following forms:

```python
X = MySqrt(2)
X = MySqrt(N)  # Not the same as the N inside the function!
X = MySqrt(float(input("Enter a positive number")))
```

Not all functions return a value. If they do something but don’t need to pass back a value, then the `return` statement will stand by itself. For example, a function that does nothing but print, or call other functions, or the main program, would look something like this:

```python
def Main():
    N = float(input("Enter a positive number"))
    X = MySqrt(N)
    print("The square root is ", X)
    return
```
Parameters

There may be as many parameters to a function as needed, but the call to that function must supply a value for each parameter, unless there are default parameters. For example, the following function may be called with 2, 3, or 4 parameters (default parameters may only appear at the right end of the parameter list):

    def MyFunction(A,B,C=0,D=1):
        # do something here

Example calls

    MyFunction(1,2,3,4,5)    # Error, too many parameters
    MyFunction(1,2,3,4)      # A=1, B=2, C=3, D=4
    MyFunction(1,2,3)        # A=1, B=2, C=3, D=1
    MyFunction(1,2)          # A=1, B=2, C=0, D=1
    MyFunction(1)           # Error, not enough parameters

The return statement is generally placed at the end of a function, but it may also appear elsewhere. For example, the following function has the return statement in two different places:

    def MyFunction(N):
        if (N < 0):
            return "Error"
        else:
            return math.sqrt(N)
The lambda expression and functions as parameters

Functions may call other functions (including themselves in the case of recursive functions), and functions may be passed as parameters to other functions. Function as parameters may be named functions or may be lambda expressions (anonymous functions). Here is a function that expects a function to be passed to it in variable FN:

```python
def Action (FN):
    N = float(input("Enter a number --- "))
    print (FN(N))
    return
```

This may be called with a predefined function such as sqrt or log from the math library (or a user defined function with one numeric argument):

```python
Action (math.sqrt)
Action (math.log)
```

However, it may also be called with an anonymous function (in this case a function that returns a value that is two times its argument plus one):

```python
Action (lambda X : 2*X+1)
```

Lambda expressions may be assigned to variables. The following two forms are virtually identical:

```python
def Special (X):
    Result = 2 * X + 1
    return Result

Special = lambda X : 2*X+1
```

Global variables

Global variables are used to pass information into and out of functions, and between functions, without going through the parameter list or return statement. Such variables are shared equally between all functions that declare them as global. Any function that declares them as global can reference them – this is why it is often a bad idea to use globals. The following two functions can communicate this way through assigning or referencing variable Q:

```python
def F1 (M):
    global Q
    return

def F2 (N):
    global Q
    return
```
The range function

The important thing about for loops is the list of values it steps through. Generating the correct lists is therefore a critical task. This is the job of the range function, to create a list:

\[
\begin{align*}
\text{range}(N) & \quad \text{generates the list} \quad [0, 1, 2, 3, 4, \ldots, N-1] \\
\text{range}(M, N) & \quad \text{generates the list} \quad [M, M+1, M+2, M+3, \ldots, N-1] \\
\text{range}(M, N, P) & \quad \text{generates the list} \quad [M, M+P, M+2P, M+3P, \ldots, N-1]
\end{align*}
\]

For example:

\[
\begin{align*}
\text{range}(10) & \quad \text{generates the list} \quad [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \\
\text{range}(3, 10) & \quad \text{generates the list} \quad [3, 4, 5, 6, 7, 8, 9] \\
\text{range}(3, 10, 2) & \quad \text{generates the list} \quad [3, 5, 7, 9]
\end{align*}
\]
Exception handling try–except and try–finally

Sometimes it is possible to check for all possible error conditions before a computation is allowed to proceed:

```python
if (N > 0):
    Result = M / N
else:
    Result = 0.0
```

This isn’t always possible, so the alternative is to proceed with the calculation as if nothing was wrong, but pick up problems that would crash the program only when and if they occur.

```python
try:
    Result = M / N  # Statement that might fail
except:
    Result = 0.0  # Statement to do if failure happens
```

There is another form as well, that is particularly helpful when writing to files:

```python
Handle = open("MyFile.txt","w")
try:
    for I in range(1000000000):
        # Statements that
        Handle.write(I,math.sqrt(I),"\n")  # might fail.
finally:
    # Error or no error
    Handle.close()  # do this anyway.
```

In this case, it is possible that the disk will fill up before the file is closed, so if that kind of error happens the file will still be properly closed. The two forms are:

```python
try:
    # statements
    # that might
    # fail
except:
    # do this only
    # if an error
    # occurs
```