# Introductory Programming in Python

March 10, 2013

# Chapter 1

# Basic concepts

## 1.1 What is a Program?

Simply put, a program is a set of instructions on **how** to take some **input** data and produce **output** data. Programs can be written in many languages, such as PERL, Python, C, Ruby, Pascal, etc... Even a stylised form of natural languages such as English, known as pseudo code, can be used to describe a program effectively.

**Input -> Program -> Output**

It is important to realise that the **program** determines what input is satisfactory, and what input will cause error. Input cannot be used to change how a program works. Similarly, the program defines what output will be produced. If you need a program to handle input differently, you need to write a new program. Likewise, if the output isn't what you need, you will need to change the program too.

Ultimately a program, no matter what language it is written in, consists of a some atomic actions/instructions, each one an instruction that cannot be divided into a sequence of 'smaller' or 'less complex' instructions. These instructions are composed (as in mathematical composition) in various manners, usually by issuing them in sequential order, but also by defining the result of one instruction as the operand of another.

Instructions (or sets of composed instructions) can be divided into two groups, those that produce a result, and those that don't. For example, adding two numbers, 3 and 4, together has a result (7). More specifically, it has a **value**, i.e. some data the program can continue to work with. Contrast this with an instruction that puts a line of text on the screen. It produces no value as a result. Instructions which produce a value are called **expressions**, instructions which produce no value are called **statements**

In a similar manner to statements, expressions can be combined with other expressions, to form complex expressions. Expressions are combined using **operators** such as the mathematical functions sum, difference, product, quotient, etc... Thus 1+1 is also an expression that has **a single value** despite the fact that it is a composite of multiple sub-expressions.

Which we can compare to the much simpler version of the same thing in python

```
a = 12
b = 12
if a*b == 144:
    a = 1
```

The basic point is to illustrate that programs are made up of small, well defined, **easy to understand** steps in sequence. As in chess, where each individual piece can only move in a very limited number of ways, yielding a tiny number of potential moves per piece per turn, these moves can be combined in a near infinite number of ways and often grouped together into common techniques or strategies. So too can the simple statements of a programming language be combined in infinite ways to produce complex but meaningful results.

## 1.2 Everyone can program!

Believe it or not, you've programmed before. You've programmed your friends, and do so every time you give them directions. After all what is a set of directions but a sequence of instructions.

```
OUT OF Cape Town TAKE the N1
TAKE the Sable rd. Offramp
AT the fork VEER LEFT
AT the traffic lights TURN LEFT
AT the NEXT traffic lights TURN RIGHT
AT the traffic circle TURN LEFT
AT the NEXT traffic circle TURN RIGHT
AT the t-junction TURN LEFT
```

You will note that some (in fact a hell of a lot) of the words in the directions to my place are capitalised. In the language of giving directions, these are pretty much our most basic instructions. That is, they are the **atomic** instructions of the program. This means that each of these instructions cannot be divided into a combination of more basic instructions For example, LEFT, on it's own doesn't make sense. Likewise, TURN alone is vague. Thus TURN LEFT would be considered an atomic instruction, which along with other atomic instructions could be used to build up a more complex program of instructions.

The portion of the directions left in lowercase are labels or names for things that are not common to all sets of directions, most often places specific to the set of directions being given. These have values, and would be our expressions.

Examining the directions we have

**OUT OF** meaning I must first be **in** a named place before performing the next instruction

**TAKE** meaning to drive along, or turn off onto a named offramp

**AT** continue until the named place or situation is reached **before** performing the next instruction

**VEER** meaning to stay in a particular lane as the road splits

**TURN LEFT, TURN RIGHT** self explanatory

**NEXT** meaning the next object of specified type encountered

At first the description of these concepts may seem obvious, but recall that computers, the machines executing your programming instructions, have an IQ of 0. They are not intelligent, fiendishly annoying at times perhaps, but never intelligent, never aware, never capable of the massive amounts of understanding and contextualization done by the human brain in order to draw logical conclusions. They are designed and built to understand and execute only specific very basic steps. So what is implicit in the instructions contained in a set of directions given to us, must be explicitly defined for a computer.

## 1.3 Data representation and translation of real world problems

Now that the concept of sequences of statements has been thoroughly flogged to death, to what do these statements apply? They apply to **data**! But data in a computer, like instructions, must be simple and well defined, or at the very least able to be broken down into multiple well defined simple pieces. In general computers work only with numbers. The pictures you see on screen, the text you are reading, the sound you hear when playing MP3s are all numbers. The actual physical devices attached to the computer are what are responsible for transforming the numbers with which the Central Processing Unit, or 'brain' of a computer, deals into humanly recognisable phenomena such as sound waves and images. Until the screen, or the speakers, are reached, everything is numbers. So it stands to reason that the most basic, atomic, unit of data in a computer is a number. Fortunately, modern programming languages are capable of dealing with numbers and sequences of numbers in a few different ways. Integers and Reals can be considered atomic data units in almost every modern computer language, as can text in the form of a string of characters in sequence.

As programs are usually written to solve problems occurring in the real world, it falls to the programmer to translate the problem being solved into something the computer can deal with, i.e. numbers. This is a bit like those annoying word problems we encountered in junior school mathematics.

> Jane has seven apples, Mary has four, Bob has one. They pool their resources, and divide the apples equally. How many apples does each one receive?

The most difficult concept to grasp when learning to program is the ability to translate a problem expressed in words into a set of instructions that describe the solution to the problem. Learning a programming language doesn't teach one to program, it merely provides one with a specific set of tools with which one can solve a problem. Learning how to apply these tools is the true skill to programming, and this comes primarily with experience. The problem set out above is ridiculously simple, and you've already worked out the answer in your head, but **how did you do it?** Describe the process! But what if there were 1000 people involved, and many thousands of apples. Working it out in your head becomes a tedious task, but the basic process you followed in your head for three people applies equally well to the case of a thousand people. And so for our first exercise in programming let us translate the word

problem into the atomic (most basic) statements and atomic data units that can be used to provide us with the answer. Assume we are provided with only the following statements and expressions to work with, and that statements are numbered from 1 upwards in the order in which they appear in our program:

- EXPRESSION – `input()`: get a number from input
- STATEMENT – `labelname = #`: Assign the value of number to a label for storage
- STATEMENT – `labelname += #`: Addition of the second number to the number stored in labelname
- STATEMENT – `if # != #` {}: check if the two numbers are not equal. If they are not equal perform any instructions within the braces {}
- STATEMENT – `GOTO #`: Instead of executing the next statement, execute the statement numbered `#`
- STATEMENT – `labelname /= #`: Division of the number stored in labelname by `#`
- STATEMENT – `print(#)`: Output of a number to the screen

Note that `#` can be either an actual number, the name of a label storing a number, or the instruction `input()` which is the number received as input.

> Each of a number of people has at least one apple. They pool their resources and divide the apples equally. For any given number of people and the number of apples each of these people has, how many apples will each person receive?

Given only the above statements and expressions to work with, there are some important questions that need answering

- Do we need to know who started with how many apples?
- How do we represent how many apples there are?
- How can one determine the total number of people?

```
1: apples = 0
2: people = 0
3: a = input()
4: people += 1
5: if a != 0 {
6:     apples += a
7:     GOTO 3
   }
8: apples /= people
9: print(apples)
```

## 1.4 Exercises

1. What is a program?

2. What is the difference between an expression and a statement?

# Chapter 2

# Invocation

## 2.1 Starting the interactive shell

Starting Python is easy.

1. We will be using IDLE to get things started
2. On the Desktop you will see a link named "Python"
3. To open the Interpreter click on **"Run" ¿ "Python Shell"**
4. The screen will open up and look similar to the following

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012,
   01:25:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more
   information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may
   be unstable.
Visit http://www.python.org/download/mac/tcltk/ for
   current information.
>>>
```

What you see now is known as the Python interactive shell. The first line tells you what version of Python you are running. The second line gives you some information about how this particular copy of Python was built, and on what system it is running. The third line lists some commands you can use to get more information. Whilst in the interactive shell, you can enter Python expressions and see their results immediately. Try it now, type in a simple mathematical expression such as `4 + 7`.

```
>>> 4 + 7
11
>>>
```

As you can see, the answer or result of the expression is printed out, and you are returned to the prompt `>>>`. Now try something different: type in `a = 2 + 7`. Then, on the next line, type just `a`.

```
>>> a = 2 + 7
>>> a
9
>>>
```

This should be somewhat familiar from the end of the basic concepts section, where we were assigning a value to a name. In this case the name is `a` and the value is the result of `2 + 7`. Note that no value is printed out immediately after the assignment. The interactive shell always prints out the value of expressions, but not of statements. This means however that the labels to which we assign values, more correctly called **variables**, are expressions that evaluate to a value, which is why it printed out `9` when we input `a`.

## 2.2 Running a saved Python Script

Whilst ideal as a calculator and for exploring new ideas quickly, it would be pretty tedious if we had to re-enter our program into the interactive shell every time we wanted to run it. So instead we can, and in fact most often will, save our program code to a file. Program source code is plain raw text. As such word processors like Microsoft Word, Wordperfect, Open Office etc... are not suitable to the task. We will look for Idle which is specifically geared towards creating Python programs, and comes with Python.

So now, instead of running the interactive shell, let's write our first Python program, this is also known as a script. Open up the fist window again and do the following

1. Click on **File** > **Save**
2. Navigate to /home/*username*/Desktop
3. Save the file as **hello.py**
4. Type the following:

```
print "Hello World!"
```

Hit F5, or Click **Run** > **Run Module** to run your program.

```
>>>
Hello World!
>>>
```

This is a very simple program, but it should help you get the idea. The computer is only following the instructions you provided to it. Lets convert the test we did on the Interpreter to a program. Save it as **second.py**.

```
#My second Python program
4 + 7 #This should add two numbers and output the
   result
```

```
a = 2 + 7
a
```

Notice the first and second lines. They contain what are called **comments**. Any text following a hash character on a line in a Python program is a comment, including the hash itself. Comments are completely ignored by Python, and are there purely to annotate code and make things easier for humans reading the code. We use comments to place small reminders within the code for ourselves, or explain the logic behind particularly tricky sections. As we progress through the course, you will find the code examples used are sprinkled more and more liberally with comments explaining how they work.

Now it would be reasonable to expect that if we ran this program we would get the same results as given to us by the interactive shell. So, let's try it.

```
>>>
>>>
```

No output? Nothing? The Python interpreter (python), when called without a file name following, starts up in the interactive shell. Only then will it output the results of expressions entered. When invoked with a file name following, and that file contains Python code, Python will only produce output if explicitly told to do so. So lets use a trick from our first program. Modify the code so it looks like the following:

```
#My second Python program
4 + 7 #This should add two numbers and output the
   result
a = 2 + 7
print a
>>>
9
>>>
```

Ah, that's better. Again, you should recognise the print statement from the end of the basic concepts section. The print statement will be explained in greater detail the next section.

One more important consideration is that Python is **case sensitive**. A variable `a` and another variable `A` are not the same, as illustrated below ...

```
#This program illustrates how Python is case sensitive

a = 3 # assign a the value 3
A = "Hi" # assign A the value "Hi"

# Check whether assigning to A has changed a
print a

# Check that A and a are still different
print "A = ", A, " and  a = ", a
```

Running this produces the output:

```
3
A = Hi   and   a = 3
```

## 2.3 Exercises

1. Start the Python interpreter
2. Try using the interpreter as a calculator
3. Write the Hello World program.
4. Write the second program.
5. Write a program that prints out your name.
6. Consider the following lines of code:

```
a = 9
b = 3
a/b
```

   (a) For each of these three lines, which are expressions and which are statements?

   (b) What will the output be if these lines are entered into the python interactive interpreter?

   (c) What will the output be if I run these lines from a file/script?

   (d) What changes need to be made to produce output when I run these lines from a file/script?

# Chapter 3

# Basic output

## 3.1 The print() Statement

The most basic statement in Python is `print()`. The `print()` statement causes whatever is between the brackets to be outputted to the screen. We've already encountered it previously, now it's time to understand how it works. Start up the python interactive shell, and let's explore. Type the following:

```
>>> print(1)
1
>>> print(173+92)
265
>>> print(173+92.0)
265.0
>>> print("hello")
hello
>>>
```

As one can see, the `print()` statement outputs the **value** of the expression immediately following it to the screen, and moves to the next line. Note that the third print statement produces slightly different output, namely the extra '.0'. This is because 92.0 and 92 are different to a computer. 92 is an integer, whilst 92.0 is a real number, or in computing terms a **floating point number** or **float** for short. The differences will be covered later.

Also of importance is the expression "hello" (note the double quotes). The value of "hello" is *hello*, and this is what is outputted to the screen by the `print()` statement. *hello* is designated as a **string** by enclosing it in quotes. Try `print(hello)` without the quotes and see what happens.

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

```
>>>
```

What's going on? Welcome to your first bug! We will soon learn to dissect and understand what all that means, but for the moment it is sufficient to understand that something has gone wrong. But what? Recall from basic concepts we were able to store values in *labels* or *variables*. Python consists of a limited set of key words that have special meaning. These key words form the list of basic (atomic) statements and expressions that python knows how to handle. Whenever python encounters a word is doesn't recognise, it treats this as a label name. It obviously doesn't recognise 'hello' as a statement, and thus treats it as a **variable**. Variables must have a value, because variables are expressions in and of themselves. But we haven't told Python what the value of hello is, hence it complains `'hello' is not defined`.

The `print()` statement is not so plain and boring as it seems. It can do a few more things that are worth mentioning. Try entering `print("Jane has", 7, "apples.")`

```
>>> print("Jane has", 7, "apples")
Jane has 7 apples
>>>
```

Of course we could just as easily use `print("Jane has 7 apples")` and get the same result. However, separating the number 7 out illustrates two important things about the `print()` statement. Firstly, we can in fact output the values of any number of expressions in a comma separated list, and secondly the outputs of each of the expressions in the comma separated list are separated by a single space each.

Finally, you will notice that the `print()` statement always ends off the line, and starts a new one. Simply leaving a comma on the end prevents this, e.g. `print("Enter your name:",)`. In summary:

- The `print()` statement prints out the *values* between the brackets and then prints a new line
- An empty textttprint() statement ends the current line and starts a new one.
- The `print()` statement can print multiple expressions if they are given in a *comma separated* list. In this case, a space is included between each expression's outputted value.
- The automatic newline outputted by textttprint() can be avoided by putting a trailing comma in the expression list. This will print a trailing space instead.
- If a textttprint() statement with a trailing comma is followed by an *empty* textttprint() statement, the trailing space is suppressed.

## 3.2 Some String Basics

Python treats all text in units called **strings**. A **string** is formed by enclosing some text in quotes. Double quotes, or single quotes may be used. There is a small limitation to this however, being that a string cannot be broken across multiple lines.

```
'this is a string of text'
"this is also a string of text"
'this string will
    cause an error, because it spans multiple lines'
```

Trying to enter the third string into the interactive shell yields

```
>>> 'this string will
    File "<stdin>", line 1
            'this string will
                              ^
SyntaxError: EOL while scanning single-quoted string
>>>
```

EOL meaning End Of Line. If we want to introduce line breaks into strings we can use two methods. The first, and simplest, is to use *triple quotes*, meaning three double or three single quotes to indicate both the beginning and the end of the string, as in

```
>>> """this string will not
...  cause an error
...    just because it is split over three lines"""
'this string will not\n  cause an error\n    just
   because it is split over three lines'
>>>
```

The immediately obvious disadvantage is that everything between the triple quotes is taken as-is, meaning the second and third lines of my string which I indented to line up with the beginning of the first line are indented in the string itself in the form of three spaces after those \n thingies. Speaking of which, what the hell are those things? Why does our string contain stuff we didn't put there? Well let's try to print the string out ...

```
>>> print("""this string will not
...    cause an error
...    just because it is split over three lines""")
this string will not
   cause an error
   just because it is split over three lines
>>>
```

Well the \ns are gone, but what were they? Strings are sequences of characters and are one dimensional. They have no **implicit** way to specify a line break, or relative position, or which characters are where relative to which other characters in the string in two dimensions, as displayed on a screen. Hence we get the second method of specifying line breaks within a string. There are special characters known as **escape characters** which mean special things inside strings. They all start with a backslash \ which escapes the following character from the string, or in layman's terms means the following character in the string is not a 'normal' character and should be treated specially. Some important escape characters are

**\n** line break or New line (n from the n in new line)

\t tab

\\ a plain backslash

You will see that because a backslash already has a special meaning, namely "treat the next character specially", we can't simply put a backslash into our string. So we escape the backslash with a second backslash, meaning we actually want a backslash and not a special character.

Finally, how do we actually put quotes inside a string since they indicate the **end** of a string. The easiest solution is to mix your quotes. If you want a single quote in a string, define the string with double quotes, e.g. `"I've got this escape thing all figured out!"`. Alternatively, you can actually escape quotes within strings, to give them the special meaning that they don't end the string, e.g. `'I\'ve got this escaped thing totally figured out!'`

## 3.3 Exercises

1. What does the `print()` statement do generally?
2. Start the Python interactive interpreter:
   (a) Output your first name.
   (b) Output your surname.
   (c) Output your first name followed by a space followed by your surname.
   (d) Create a variable called `firstname` and put your first name into it.
   (e) Create a variable called `surname` and put your surname into it.
   (f) Using only the variables you have created, print your first name and surname again, making sure there is exactly one space between your two names.

   Quit the Python interactive interpreter.
3. Which special rules does the print statement adhere to regarding trailing spaces and newlines?
4. Consider the following code...

```
print("MUCH madness is divinest sense,")
print("To a discerning eye;")
print("Much sense the starkest madness.")
print("'T is the majority","In this, as all,
   prevails")
print("Assent, and you are sane;")
print("Demur,-you're straightway dangerous",)
print()
print("And handled with a chain.")
```

(a) What output, exactly, does the above code produce? Indicate spaces with underscores.

5. Write a program that prints "Hello".
6. Write a program that outputs a favourite piece of poetry or other prose, over multiple lines.
7. How can you print a value stored in the variable `x`?
8. How can you print the values of multiple expressions on one line?
9. Write a program that outputs your name, age, and height in metres in the following format. Make sure age is an integer, and height is a float, and not simply part of your string.
   `My name is James, I am 30 years old and 1.78 metres tall.`
10. Explain three possible ways to print a string containing an apostrophe, for example the string
    `The cat's mat.`

# Chapter 4

# Basic Input

## 4.1 Literal Values

So far our programs have been pretty uninteresting. They consistently produce the same results because they consistently act on the same data, or input, which we have specified by *hard coding* values for our inputs in the form of **literals**. A constant or literal is sort of the opposite to a variable, in that its value does not change over the course of program execution, and more importantly, its value is specified *literally*, within the program code.

As mentioned before, Python understands a fairly limited set of key words and symbols as basic statements or operators. Previously, we said that Python treats unrecognised words (used is a very loose sense) as variable names, but this is not strictly true. Python treats unrecognised words as expressions, and attempts to determine their value. Now, the word can specify a value directly, in which case it called a literal value, or simply *literal* for short.

```
#An example differentiating literals and variables
a = 3                  #a is a variable name, 3 is an
   integer literal
b = 'b'                #b is a variable name, 'b' is a
   string literal
c = True               #c is a variable name, True is a
   boolean literal
if 3*'b' == a*b:       #3 and 'b' are literals, a and b
   are variables
    print(c)           #is c literal or variable?
else:
    print "False"     #is "False" a literal or a
       variable?

#Are False and "False" the same?
```

## 4.2 The input() Function

It's not very helpful to us if we must change our programs every time we want to change the data they work with. So instead we want to be able to tell our program to get input from somewhere. The simplest place to get input from is the keyboard, in the form of text entered by the user. Python provides a **function** to do just this, called `input()`:

```
>>> name = input("What is your name? ")
What is your name? James
>>> print(name)
James
>>>
```

We've seen *input* before in the basic concepts section, and there really isn't much to it. Just a few things to note. Firstly, input is a function. A function is a defined collection of statements that produce or **return** a value when executed. You could think of them as a way of turning a small set of statements into an expression. We will learn how to define our own functions later on in the course, but Python provides quite a few basic built in ones which we are going to be using before that, so let's look briefly at how they work.

- `input` is the name of the function. Like variables, functions have names to identify them. Like a variable, a function name is a label that points to the collection of statements to be executed.
- To execute the statements in a function, better known as **calling a function** we write the function's name followed by round brackets. `input()`
- Using the name just by itself treats the function like a variable.
- `"What is your name? "` is a parameter to the function. Functions are like mini programs. They always do the same thing, but they can do the same thing to different data. We put the data we want a function to operate on in between the brackets when we call it. Expressions and variables **passed** to functions in this way are temporarily known as **parameters**. Multiple parameters can be passed to a function by separating them with commas, and in general any valid expression is a valid parameter; e.g.
  `minimum(6, 2, 3+4, 4**2, 8.3, 3*1)`
  has six parameters, some of which are not just simple expressions like plain numbers.
- Despite the fact that any valid expression is a valid parameter, functions are usually quite picky about the parameters they can work with. For example, we couldn't find them minimum of a collection of strings. That only really works with integers and floats. Functions usually specify the *number* of parameters they accept. Often functions may accept less parameters than they specifically ask for, by substituting default values for the parameters not provided.

Back to `input`! `input()` captures a single line of text from the keyboard, as entered by the user. It returns a string containing the captured text, leaving out the `\n` produced when the user hits the Enter key. In addition, it takes one optional parameter, which is displayed as

a prompt prior to accepting input from keyboard. If this parameter is left out, the default prompt is an empty string, so nothing is printed.

## 4.3 Exercises

1. If there is a function named *my_function*, how do I call it?
2. Are functions expressions or statements? What about when they are called?
3. Start the python interactive interpreter:
   (a) Assign what the user types to a variable called `s`.
   (b) Print the value of the variable `s`.
   (c) Print `s`.
   (d) Use `input()` to prompt the user for a number, get that number, and assign it to a variable called `n`.
   (e) Print double the value of the variable `n`.

   Exit the Python interactive interpreter.
4. Write a program that asks the user for their name and stores it in a variable. Then output "Hello", followed by the user's name
5. Write a program that asks the user to enter two numbers, and prints the sum of those two numbers.
6. Write a program that asks the user to enter two numbers, and prints the difference of those two numbers.
7. Write a program that asks the user to enter two numbers, and prints the product of those two numbers. Are you sensing a pattern here?
8. Write a program that asks the user for some text, and a number. The program prints out the text a number of times equal to the number entered, without line breaks or spaces in between each repetition.

# Chapter 5

# Program state

## 5.1 The Concept of State

As we know, programs execute statements in sequential order. They flow through your code, left to right, top to bottom. As statements are executed, things happen: output is issued, calculations are done, and the **state** of your program changes. But what is state? Loosely put, state refers to all the data your program is dealing with and their **current** values at the point in time when a particular statement is about to be executed. Since the entire point of a program is to transform input data into some meaningful output data, it stands to reason that the program's state will change as it executes statements. We need a way to record the state of our program in manageable units. We shall call these units **variables**, since their values may vary as our program executes.

Variables each have a name, a type, and a value. A name is simply what we choose to call a particular piece of data. For example, we previously called the bit of data representing the total number of apples held by everyone 'apples'. We stored a number in 'apples', so this would be its type. The value of 'apples' started at 0. As more people were added to the problem, the number of apples increased. Accordingly, the value of our variable 'apples' changed.

**Type** refers to the type of value a variable can have. Text is not the same as a number, thus they are considered different types. One could not for example add "apples" to the number 7.

Python has five basic types:

**int** Integers, e.g. 1, 179835646, -3, 0

**string** Text strings, e.g. 'Alice', "The cat sat on the mat"

**float** Real numbers, e.g. 0.0, 48747.23501, -0.5

**bool** Boolean conditions, e.g. True, False

**None** None is used to specify a variable which has no type or actual value.

## 5.2 Assignment Statements

To create a new variable, we simply **assign** it a value, by giving it a name and setting the name equal to the value. We do this by using the **assignment statement** which takes the form `variable_name = expression`

```
>>> name = "James"
>>>
```

Here we have created a new variable, whose name is `name`, and which has the value *James*. Note, we did not have to say that `name` was of type string. When assigning a value to a variable, Python automatically sets the type of variable being assigned to the type of the value being assigned. If we go on to assign name again, this time with a different value

```
>>> name = "Jimbo"
>>>
```

the value of `name` will be changed. If we want to see what the value of a variable is we can simply print it:

```
>>> print(name)
Jimbo
>>>
```

James has disappeared. As far as the program is concerned James was never there. There is no James! Assigning a new value to a variable will obliterate the previous value of that variable. If you wish to keep the old value of a variable you need to save a copy, in a different variable.

```
>>> i = 1
>>> j = i
>>> i = 2
>>> print("i =", i, "and j =", j)
i = 2 and j = 1
>>>
```

Let's look at what we've done, step by step.

1. `i = 1`
   We create a new variable, called `i`, assign it a value of *1*, and Python sets it's type to `int`
2. `j = i`
   We create another new variable, this time called `j`, assign it a value of ... well what exactly? `i` is not a string because it's not in quotes, and it's not a number, so it's being treated as a variable. But what is `i`? Variables are implicitly considered to be their current values. So `i` is implicitly considered to be 1 (it's current value)
3. `i = 2`
   We assign a new value to `i`. The old value of `i` is discarded.

4. `print("i =", i, "and j =", j)`
   We output the current values of `i` and `j` respectively.

Step 2 highlights a number of important points about assignment. The assignment statement does something very specific. It changes the current value of the variable being assigned to the value of the expression to the right of the equals. This means that the value of the expression is determined before actually changing the value of the variable. As you can see from the output produced by our little program, assignments do not behave in the same way as a similar statement in maths does. `j = i` does not mean that `j` and `i` are always equivalent. The only thing that can be certainly said about the relationship between `j` and `i` is that they will be equal directly after the assignment statement has been executed. Both before and after that point in time, all bets are off. An assignment is **not** a description of a relationship between two expressions.
It effects a **once off** change in the value of one variable.

Consider the assignment statement `x = x + 1`. For those of you with inner mathematicians that are screaming loudly to get out, prepare yourself! What we have is an assignment to a variable (`x`) the value of an expression (`x + 1`). Let us suppose that currently the value of `x` is 1. This assignment occurs as follows.

1. `x = x + 1`
   The value of the expression `x + 1` is determined. Since `x` is 1, we can transform the expression into `1 + 1` since variables in expressions are considered to have their current value.
2. `x = 1 + 1`
   This is obviously 2, hence the value of the expression overall is 2.
3. `x = 2`
   2 is assigned as the **new** value of `x`.

## 5.3 Integers, Floats and Arithmetic Expressions

Obviously we want to be able do more than simply assign values to variables. We need to be able to manipulate and combine them in various ways. For integers and floats this leads directly to arithmetic notation, and its representations in Python. In general, basic arithmetic expressions can be formed in much the same way as one would express them mathematically. If `x` and `y` are either integers or floats then

- `x + y` yields Addition
- `x - y` yields Subtraction
- `x * y` yields Multiplication
- `x / y` yields Division
- `x % y` yields Modulo (or Remainder)
- `x ** y` yields Exponentiation (or raising to the power of)

The above list describes various **operators** that can be used to form arithmetic expressions. But, suppose we have the expression `2 + 3 * 4`. Which of the addition or the multiplication operators is applied first? One might think the operators are applied simply left to right, but as in maths, we have a well defined order of **precedence** specifying the order in which operators are applied. In general Python arithmetic operates exactly as it would in normal mathematics.

We know that variables have a type, and so does the value of expressions. The type of the value of an expression is generally determined by the types comprising the expression. For example, adding one integer to another in an expression always yields an integer, so the expression's type is integer. But what happens when you mix two, or more, types in an expression? In general, all participating sub-expressions have their type promoted to the most general type. By general, we mean *most capable of representing all types involved*. For example, the value 0.5 cannot be represented by an integer, but the value 3 can be represented by a float as 3.0. Hence adding a float to an integer means the integer gets promoted to a float and the type of the expression's value as a whole will be a float.

Python division allows us to do a number of things. We can divide two integers (say 5/2) and the result will be represented as a float (2.5). However, we often actually want the integer part of a division only (that is, we want the decimal part dropped), or the remainder. Python provides us with an operator to get both the integer division value and the remainder. Using % is called getting the modulo of two numbers:

```
>>> print(5 % 2)
1
>>> print("5 divided by 2 is ", 5 // 2, "remainder", 5
   % 2)
5 divided by 2 is 2 remainder 1
>>>
```

## 5.4 Strings and common operators

Strings can be manipulated too, but the operations we perform on them are fundamentally different. Assuming `s1` and `s2` are strings, and `i` and `j` are integers

- `s1 + s2` yields Concatenation
  This joining of two strings in the order they are listed as operands to from a single string; e.g. `"a" + "b"` yields `"ab"`
- `s1 * i` yields Repetition
  The creation of a new string repeated $i$ times; e.g. `"a" * 3` yields `"aaa"`
- `s1[i]` yields Subscription (or indexing)
  The extraction of a single character at a specific position in the string, where the first character in the string is a position 0, the second and position 1, and so on; e.g. `"abc"[1]` yields `"b"`. $i$ may be any expression evaluating to an integer. Negative numbers may be used, where `-i` means the i'th last position.

- `s1[i:j:k]` yields Slicing
  The extraction of a series of characters from a string starting at the character in the i'th position and continuing until the character just prior to the j'th position, extracting only every k'th character, where positions are numbered from 0, and negative numbers mean distance from end of string. Any of i, j, or k may be omitted, in which case i is treated as 0, j as the string length, and k as 1. A k value of 1 means all characters in the specified range are extracted. If j is omitted, the extraction is from the i'th position to the end of the string. Examples:
  `"The quick brown fox"[4:]` yields `"quick brown fox"`
  `"The quick brown fox"[-4:]` yields `" fox"`
  `"The quick brown fox"[:4]` yields `"The "`
  `"The quick brown fox"[4:9]` yields `"quick"`.
  `"Thequickbrownfox"[::2]` yields `"Teqikbonfx"`
  `"The quick brown fox"[4:15:3]` yields `"qcbw"`.

- `s1[:]` yields Slicing of the entire string, essentially making a complete copy of the string.

## 5.5 Formalisation of the Concepts of Statements and Expressions

To recap:

1. Programs consist of sequences of statements
2. Statements perform actions but do not have value
3. Statements may act on expressions
4. Expressions have a value, and when used in statements, their value is determined and substituted in place of the expression
5. Expressions may be simple expressions such as numbers or variables whose values can be determined directly
6. Expressions may be composed of simpler expressions combined using appropriate operators

## 5.6 Exercises

1. What does the assignment statement do?
2. What are the five basic variable types in Python?
3. If the integer variable `i` has the value 7, then what is the value of the expression `7/4`. Why?

4. How does one calculate the remainder (modulo) of an integer when divided by another number?
5. Write a program that asks the user for a number from 1 to 31. Assume the first day of the month is a Sunday. Output the name of the weekday of the day of month the user entered.
6. Write a program that outputs the letter "x" 1000 times on a single line, without intervening spaces.
7. If `s` is a string variable with the value `"Harry's Hippie Hoedown"`, then what is the value of
   `s + ":  tickets only $5"`
8. If `s` is a string variable with the value `"Harry's Hippie Hoedown"`, then what is the value of
   `s + ":  tickets only $" + "5"*3`
9. What is the value of `"ABBA was a Swedish band popular during the 80's"[0:4]`?
10. What is the value of `"ABBA was a Swedish band popular during the 80's"[-15:-7]`?
11. If the string variable `s` has the value `"ABBA was a Swedish band popular during the 80s"`, then what is the value of
    `"BAAB"+s[4:11]+"Danish"+s[18:24]+"un"+s[24:-4]+"90's"`
12. What is your favourite sport? Write a program that inputs the score units for this sport (e.g. for rugby: number of tries, conversions, penalties, time played) and works out some statistics (e.g. the total score, average points per minute, time remaining, expected final score).