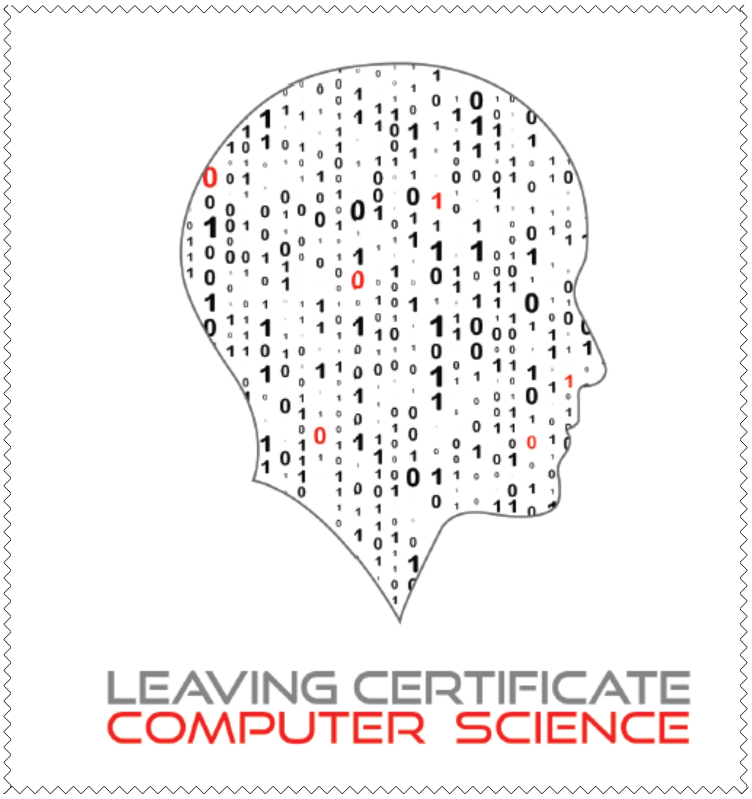
LC CS Python

Student Manual



Section 2

Data, Variables, Assignments and Expressions

Name: ___________________________

Introduction

So let's say we wanted to write a program to display the name of a person and their favourite colour in a greeting string and then displays a personalised goodbye message. We could write:

```
print("Hi Alex - your favourite colour is red")
print("Goodbye Alex")
```

One problem with the above program is that the string *Alex* appears twice and this gives rise to the possibility of a mismatch in spelling. It would be better if had some way of telling our program to remember the person's name. This can be done by using a *variable*.

KEY POINT: A variable is a programming construct used to store (remember) data.

The listing below uses two variables – `personName` and `favouriteColour`.

```
1. personName = "Alex"
2. favouriteColour = "red"
3. print("Hi", personName, "- your favourite colour is", favouriteColour)
4. print("Goodbye", personName)
```

The variable personName is used to store a person's name and the variable `favouriteColour` is used to store the person's favourite colour.

The variables are declared on lines 1 and 2 respectively. Each line assigns the initial values *Alex* and *red* to the respective variables.

KEY POINT: A variable must be declared before it can be used. By declaring a variable you are telling Python *here is a new word and this is its initial value*.

Line 3 displays the contents of the variables in a greeting string. Notice that the names of the variables appear outside the double quotations, and also the use of commas to delimit the variables from the greeting string.

When Python comes across the variable names in the `print` command it substitutes the values of the variables into the string to be displayed.

The name of the person or the colour can now be changed, simply by changing the value of the variable. For example, we could write:

```
personName = "Charlie"
favouriteColour = "green"
print("Hi", personName, "- your favourite colour is", favouriteColour)
print("Goodbye", personName)
```

The program is considered better because the person's name is stored in a variable and needs to be keyed in by the programmer only once. However, the program still has a problem in that it lacks generality i.e. it only works for one person and one colour. Every time we want to display a different message we need to change the program.

- In handout, do Task 1.

A more general (and realistic) solution would be to ask the user to enter their name and favourite colour. This can be achieved using the `input` command as follows:

```
personName = input("Enter your name: ")
favouriteColour = input("Enter your favourite colour: ")
print("Hi", personName, "- your favourite colour is", favouriteColour)
print("Goodbye", personName)
```

The `input` command

The `input` command allows a user to enter a value into a running program and have that value stored in a variable.

The string in brackets following the word `input` is displayed as a prompt to the end-user.

Every time the above program is run, whatever values are entered by the end-user are stored in the variables `personName` and `favouriteColour`. These values are then displayed in the output messages.

Without having to make any changes to the program, the output messages can vary on every run. This is an example of a general solution to a problem.

The `input` (and `print`) commands are both examples of Python *built in functions*. Other examples include `int`, `float`, `open` and `str`.

Variable Syntax

The general syntax for declaring a variable is made up of a left hand side and a right hand side as follows:

```
<variable-name> = <expression>
```

The name of the variable appears on the left hand side and an expression appears on the right hand side. The `'='` symbol in the middle is the Python *assignment operator*.

KEY POINT: Although the symbols used to denote the Python assignment operator and a mathematical equation are identical, they should not be confused as they mean two completely different things.

The use of `'='` in Python indicates an *assignment statement*. When Python comes across an assignment statement it evaluates the expression on the right hand side first. The result of this evaluation is then stored in the variable named on the left hand side.

The expression on the right hand side can be:

- a literal value such as a string or a number
- an arithmetic expression (which itself can contain variables)
- the name of a built-in command such as `input`, as seen in the previous example.

STUDENT TIP
It is useful to think of a variable as a 'box' in the computer's memory (i.e. a memory location) where a value is stored.

Every time a value is assigned to a variable that value is stored at that variable's memory location.

Once a variable has been declared the name is added to Python's vocabulary for the remainder of the program.

- In handout, do Task 2.

STUDENT TIP
Variables are used by programs to store data.

A programmer should decide to use a variable whenever they want their program to remember a value that will be needed at a later stage in that program.

When a variable is declared for the first time it must be given some initial value. This is called *initialisation*.

It is up to the programmer to decide what name to give their variables. The rules and guidelines for naming variables are described below.

Guidelines and Rules for Naming Variables

As a general guideline variable names should be simple and meaningful. A meaningful name is one that tells something about what the variable is used for. The use of meaningful variable names makes programs more readable and understandable to fellow programmers.

When choosing a name for a variable it can be helpful to think of a noun that describes the purpose of the variable.

It is considered good practice to capitalise interior words in multi-word variable names. This usage is referred to as *camel case* and `firstName`, `addressLine1`, `stockCount`, `highScore`, and `payRate` are all examples of good variable names.

The syntax rules for naming variables are as follows:

- **A variable name cannot be a word already reserved by Python (e.g. “`print`” “`def`”, etc.)**
- **Variable names must contain only letters, digits, and the underscore character, _.**
- **Variable names cannot have a digit for the first character.**
- **Spaces or dots are not allowed in a variable name**

If Python comes across a name it does not understand it will display a syntax error.

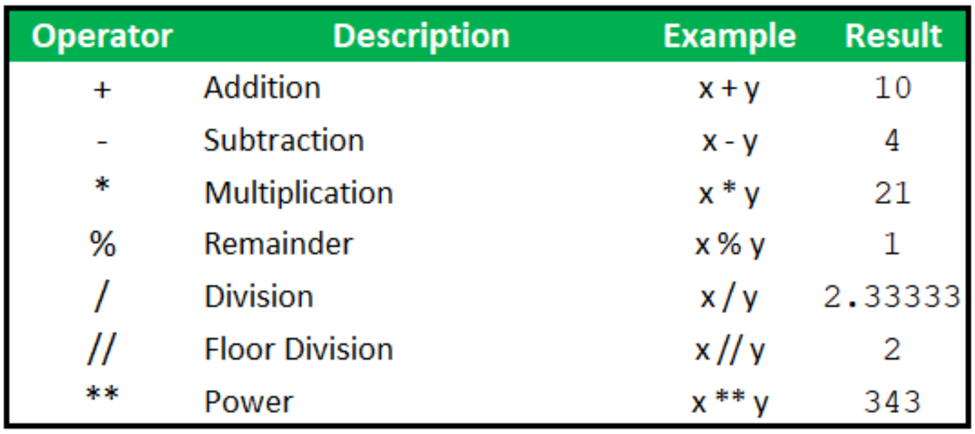
Datatypes and Operations

Programmers need to be aware of the type of data that their programs process. This is referred to as datatype.

So far, we have encountered examples of both string and numeric datatypes. If, for example, we wanted a program to store someone's name or favourite colour the variable's datatype would be string. On the other hand, a numeric datatype is the proper datatype for a variable to store a person's age or height.

Python supports several different types of numbers - *integers*, *floating point numbers* as well as a range or more exotic types of numbers (e.g. complex numbers, fixed precision decimals and rational numbers)

Every datatype in Python has a permissible set of operations that are only valid for that type. (For this reason, Python is said to be a *strongly typed* language.) The numeric datatype supports all the usual arithmetic operations such as addition, multiplication etc. These are illustrated in the table below (assume **`x=7`** and **`y=3`**)



Operator	Description	Example	Result
+	Addition	x + y	10
-	Subtraction	x - y	4
*	Multiplication	x * y	21
%	Remainder	x % y	1
/	Division	x / y	2.33333
//	Floor Division	x // y	2
**	Power	x ** y	343

Python arithmetic operators

The normal precedence rules for arithmetic operators apply.

- In handout, do Task 3.

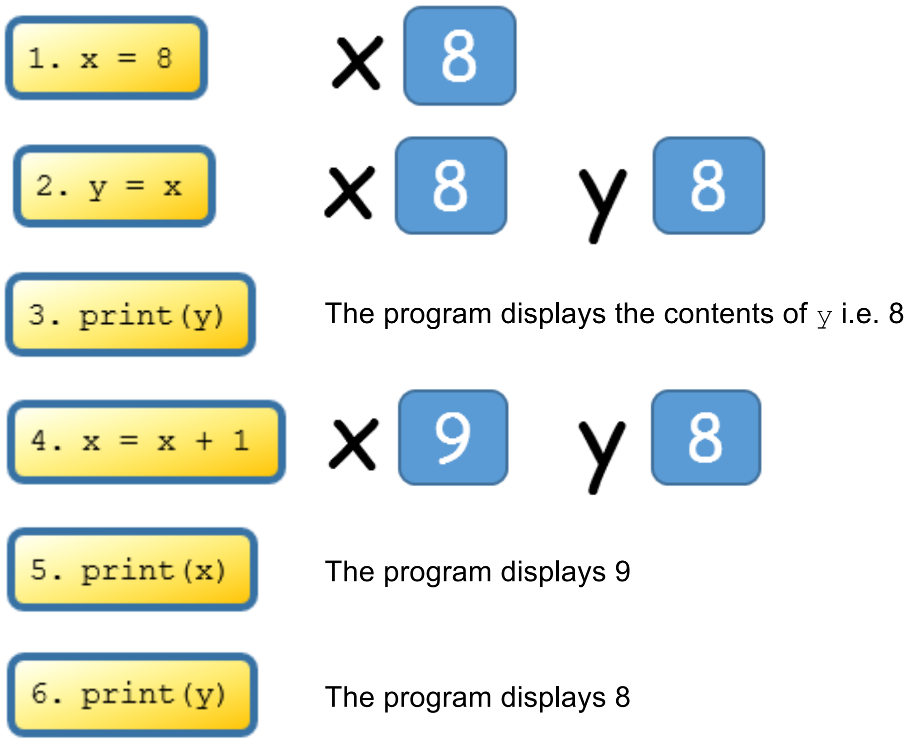
Program Tracing

Computers can execute programs at a rate of millions of lines per second. The values of variables can change so fast that programmers can easily lose track and sometimes find it difficult to be sure that their program logic is correct. In order to combat this, programmers often execute a program manually i.e. using pen and paper to keep track of variables line-by-line. This activity, called program *tracing* is used by programmers to verify for themselves that their program will do what it is intended to when it is run by the computer.

We trace through the program shown (as if we were the computer) line-by-line, starting at line 1. Every time a variable is declared for the first time we draw a box and write the value of the variable in the box. When the value of a variable changes we replace the old value with the new one. This activity is called program tracing.

```
1. x = 8
2. y = x
3. print(y)
4. x = x + 1
5. print(x)
6. print(y)
```

The Python code is shown on the left below and the variables are illustrated as boxes on the right.

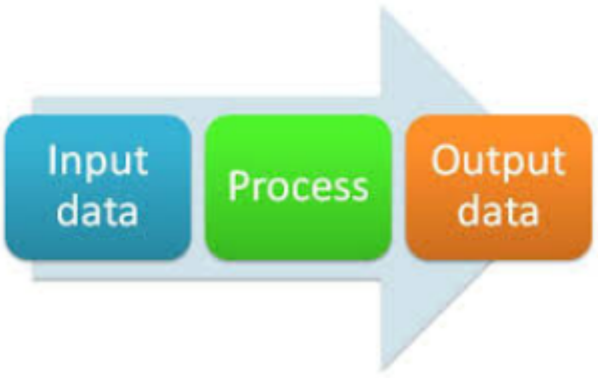


`1. x = 8`	x 8
`2. y = x`	x 8 y 8
`3. print(y)`	The program displays the contents of `y` i.e. 8
`4. x = x + 1`	x 9 y 8
`5. print(x)`	The program displays 9
`6. print(y)`	The program displays 8

- In handout, do Task 4.

Input-Process-Output

Many computer programs follow the input-process-output model as illustrated.



This means that a program accepts data as input, carries out some processing (usually a calculation) and then displays and/or stores the output.

We already know the `input` command is used to prompt an end-user to enter a value into a running program. The value entered can then be stored in a variable.

KEY POINT: programmers need to be acutely aware of the type of data with which their program is working.

By default, the `input` command returns a string. This means that if you want your program to accept numeric data from the end-user, the value entered must be converted from a string to either an integer or a floating point (i.e. a decimal) number.

Fortunately, Python has two built-in commands that can perform these conversions. These are called `int` and `float` respectively.

Built-in Function	**Description**
`int(s)`	Converts the string 's' to an integer. The result is a new number object
`float(s)`	Converts the string 's' to a floating point (decimal) number. The result is a new floating point object

The two commands `int` and `float` are important because they allow Python to use values entered by the end-user in arithmetic expressions.

Example – Year of Birth

We want to write a program to calculate a person's year of birth.

Our program will ask the end-user for two pieces of information - the current year and the age they will be at the end of the current year.

We will store this data in two variables – `year` and `age`.

Since `year` and `age` are both numeric we will need to instruct the program to convert them from strings to integers. This can be done with the `int` command.

The solution is as follows.

```
1. year = int(input("Enter the current year: "))
2. age = int(input("What age will you be at the end of this year : "))
3. print("You were born in", year-age)
```

Lines 1 and 2 both display a prompt asking the user to enter values and then convert these values from strings to integers. The conversion from string to integer is needed here because Python knows how to subtract numbers but cannot subtract strings.

Line 3 subtracts the two integers (to calculate the year of birth) and displays the result in an output message.

Notice how both `int` and `input` are called on the same line. When commands are combined together on the same line like this it is called *function composition*. Python executes the innermost function first and then works back towards the leftmost function which is executed last.

It is important to understand the subtle difference between the string "2018" and the numeric value 2018. One difference is subtraction is supported for numbers but not for strings

KEY POINT: The operations that can be carried out on values are constrained by the value's underlying object type.

Syntax Check: Try running the following program:

```
year = input("Enter the current year: ")
age = input("What age will you be at the end of this year : ")
print("You were born in", year-age)
```

You will see an error like this:

```
Traceback (most recent call last):
  File "C:\PDST\Python Workshop\src\year of birth.py", line 3, in <module>
    print("You were born in", year-age)
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Since the values are not converted to integers Python stores them as strings. The expression `year-age` on line 3 is an attempt to subtract two strings which is not allowed in Python. Python does not support the subtraction operation on strings.

- In handout, do Task 5.

It is also worth noting that numbers can be converted to string objects using the `str` function. As an experiment try running the following lines of code separately:

```
print("2000"+18)
```

This causes the following error to be displayed
`TypeError: must be str, not int`

```
print("2000"+str(18))
```

This causes the string “200018” to be displayed.

KEY POINT: The ‘+’ operator for strings means concatenation.

Example – Temperature Conversion

Let's look at another example of a program that accepts an input, performs some processing and displays an output. The program below converts Centigrade to Fahrenheit. The formula used is,

$$f = \frac{9}{5} \times c + 32$$

The input is the Centigrade value entered by the end-user; the processing is the conversion (done by applying the formula); and the output is the Fahrenheit value displayed

```
1. # A program to convert from Centigrade to Fahrenheit
2. centigrade = float(input("Enter the Centigrade value: "))
3. fahrenheit = 9/5 * centigrade + 32
4. print(centigrade, "degrees C equals", fahrenheit, "degrees F")
```

The use of the `float` command on line 2 above means that the user can enter decimal values for Centigrade. (Integer values are also permitted.)

Notice how both `float` and `input` are combined together on the same line.

A note on Testing

The purpose of testing is to verify that the program does what it is meant to.

It is normal practice for programmers to devise a number of *test cases* as part of the program design process. Each test case is made up of an input and an *expected output*.

When the test case is run the *actual output* should be recorded. If there is a difference between the expected and actual output, then the program contains an error (or bug) which will need to be fixed.

The table below provides a good basis for testing the Centigrade to Fahrenheit program shown on the previous page. Each row in the table is a separate test case.

Sample Input (℃)	Expected Output (℉)	Actual Output
0	32.0	
32	89.6	
1000	1832.0	
-10	14.0	
-40	-40.0	
-1000	-1768.0	

Note that the values in the sample input column are arbitrarily chosen. The expected output values are calculated by using a calculator or come from some other source e.g. world wide web. The values in the actual output column should be recorded by running the program.

When the program passes all test cases it is said to be *unit tested*.

A good unit test will ensure that every line of code is triggered. It will also take ‘abnormal’ scenarios into consideration

STUDENT TIP
Students should be encouraged to get into the habit of testing their code as early as possible in the learning process.

- In handout, do Task 6.
- In handout, do Task 7.

More built-in functions

Let’s take another look at our temperature conversion program discussed earlier.

```
1. # A program to convert from Centigrade to Fahrenheit
2. centigrade = float(input("Enter the Centigrade value: "))
3. fahrenheit = 9/5 * centigrade + 32
4. print(centigrade, "degrees C equals", fahrenheit, "degrees F")
```

This time try an input value of 21℃. The program displays the output shown below – the number of digits displayed after the decimal point is unnecessary and unwieldly.

```
21.0 degrees C equals 69.80000000000001 degrees F
```

The level of precision displayed can be controlled by the programmer by using the `round` function. Line 4 tells Python to display the value of `fahrenheit` rounded to 1 decimal place.

```
4. print(centigrade, "degrees C equals", round(fahrenheit,1), "degrees F")
```

A full description of the `round` function is given in the table below along with some other useful build-in functions offered by Python.

Function	Description	Example(s)

`round(x [,n])`	Rounds the number `x` to `n` fractional digits from the decimal point. If `n` is not provided it is taken to be zero.	`round(27.168459, 1) -> 27.2`
		`round(27.168459, 2) -> 27.17`
		`round(27.168459, 3) -> 27.168`
		`round(27.168459, 4) -> 27.1684`
		`round(27.168459, 5) -> 27.16846`
		`round(27.168459, 6) -> 27.168459`

`abs(x)`	Returns the absolute value of x	`abs(-273)   -> 273`
		`abs(-1.27)  -> 1.27`
		`abs(0)      -> 0`
		`abs(32)     -> 32`

`pow(x, y)`	Calculates `x` to the power of `y`. (Same as `x**y`)	`pow(5, 2)   -> 25`
		`pow(2, 16)  -> 65536`
		`pow(4, 3)   -> 64`

The Remainder Operator (%)

The remainder operator, % (aka modulus operator), like all of the other Python arithmetic operators, is a binary operator i.e. it works on two numbers, referred to as operands.

The comments (in red) short Python program below describe how examples of the remainder operator works.

```
print(30 % 10) # displays 0 because 30 divided by 10 leaves no remainder

print(30 % 15) # displays 0 because 30 divided by 15 leaves no remainder

print(30 % 20) # displays 10 because 30 divided by 20 leaves a remainder of 10

print(9 % 4)   # displays 1 because 9 divided by 4 leaves a remainder of 1

print(9 % 5)   # displays 4 because 9 divided by 5 leaves a remainder of 4

print(12 % 5)  # displays 2 because 12 divided by 5 leaves a remainder of 2

print(5 % 12)  # displays 5 because 5 divided by 12 leaves a remainder of 5
```

It should be evident that the remainder operator works by dividing the second operand into the first. Whatever is left over is the result of applying the remainder operator.

The word *mod* (short for modulus) is often used to phrase questions or answers involving the remainder operator. For example, what is 30 mod 10, or 9 mod 5 is equal to 1. Furthermore,

- $a\ mod\ a\ is\ zero$ because any number divided by itself leaves no remainder
- $a\ mod\ 1\ is\ zero$ because 1 divides every number evenly
- $a\ mod\ 0\ is\ undefined$ because division by zero is not defined. In Python any attempt to divide by zero will always result in a runtime error.

Computer Science contains a rich set of problems whose solutions involve using the remainder operator. The application of the remainder operator to solve problems is referred to as modular arithmetic and because modular arithmetic is particularly useful for calculations involving time, it is also referred to as *clock arithmetic*.

For example, let's say wanted to find out what day of the week it will be in 1000 days' time. We know every week has 7 days, and can calculate 1000 mod 7 to be 6. Therefore, the day of the week in 1000 days will be the same as day as it will be 6 days from now.

The same general principal can be applied to working solutions to problems involving other units of time such as seconds, minutes, hours, months, years, leap years, Easter etc.

Let's take another example. You are about to embark on a space journey lasting 850,000 hours! Take off time is exactly 21:00. What time will it be when you reach your destination?

```
arrivalTime = (21 + 850000) % 24

print("You will arrive at", arrivalTime)
```

Modular arithmetic is also useful in situations where it is required to group 'things' (e.g. people) into a fixed number of arbitrary sized groups. For example, if we had a group of 20 students in a class and we wanted to create four arbitrary sized groups, we could ask each student to pick a random number – let's call it N. Then `N mod 4` will guarantee a group number for that student because it will always be 0, 1, 2 or 3.

Another useful application is simply to find the properties of numbers. For example, `mod 2` is frequently used in computer programs to test whether a number is even or not (*evenness test*).

Perhaps, some of the most common applications of modular arithmetic can be found in the area of coding systems, ciphers and cryptography. Ubiquitous examples exist in the use of modular arithmetic to perform validity checks on barcodes, ISBN numbers and credit card numbers (e.g. Luhn's algorithm) to name just a few. For example, a 13-digit barcode is valid only the following expression is evenly divisible by 10.

$$(d1 + d3 + \cdots + d13) + 3\times(d2 + d4 + \cdots + d12)$$

where, $d1$ is the leftmost and $d13$ the rightmost barcode digit. If the expression `mod 10` is not equal to zero, the barcode is invalid and the scanned item will be rejected.

Running Totals

Running totals are needed so often in programs that it is well worth putting some effort into understanding the pattern used to create them.

A running total is a value that usually starts at zero and increases by successive additions until a final total is reached. A very common example is a shopping basket total such as those calculated at a checkout.

Let’s say we have three items in our basket and they are valued at €10, €14 and €6 respectively. With very little effort, most people understand that the total bill is €30. However, what most people probably don’t realise is that they have (subconsciously) run a running total program similar to that shown below in their own heads.

```
1.  # Program to calculate a running total
2.
3.  # Initialise the variable
4.  runningTotal = 0
5.
6.  # Perform the calculations
7.  price1 = 10
8.  runningTotal = runningTotal + price1
9.  price2 = 14
10. runningTotal = runningTotal + price2
11. price3 = 6
12. runningTotal = runningTotal + price3
13.
14. # Display the output
15. print("Total amount is", runningTotal)
```

Novice programmers may find the following tips for dealing with running totals useful:

1. Recognise the need for a running total (this is probably the most difficult step)
2. Initialise your running total variable to zero (line 4 above)
3. Every time you need to add a value to the running total use an assignment (lines 8, 10 and 12 above). Notice the only difference between these lines is the name of the variable being added to `runningTotal`.

KEY POINT: The hallmark of the running total pattern is the `runningTotal` variable appears on both sides of the assignment statement. In this way it is used to update itself.

- In handout, do Task 9.

Introducing Random Numbers

Random numbers provide a rich way of generating numeric data in early stage programming. A more advanced application of random number generation is in games programming.

The following two programs demonstrate random number generation.

The first multiplies two randomly generated numbers and displays the result; the second computes the mean of five randomly generated numbers.

```python
# Program to multiply two randomly generated numbers
import random

num1 = random.randint(1,10) # generate a number between 1 and 10
num2 = random.randint(1,10) # generate a number between 1 and 10

# Multiply the two numbers and display the result
print(num1, "times", num2, "=", num1*num2)
```

```python
# Program to average five randomly generated numbers
import random

low  = random.randint(1,100)
high = random.randint(low,100)

# Generate the 5 random numbers between low and high
n1  = random.randint(low, high)
n2  = random.randint(low, high)
n3  = random.randint(low, high)
n4  = random.randint(low, high)
n5  = random.randint(low, high)

# Compute their average
average = (n1+n2+n3+n4+n5)/5

# Add the five numbers and display the result
print("The average of", n1, n2, n3, n4, n5, "is", average)
```

Additional Notes

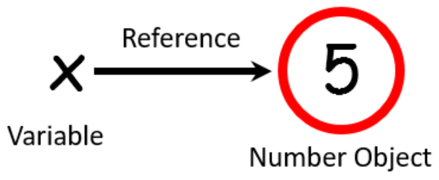
1. Variables and Memory

When a variable is assigned a value for the very first time it is said to be *initialised*. We say a variable is initialised to a value. Memory for the variable is *allocated* at runtime when the variable is initialised.

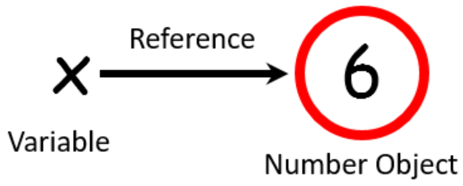
Internally, Python maintains a *system table* with one entry per variable. The values are represented in memory as objects and a reference links the variable to the object.

KEY POINT: In Python variables are actually references to objects.

The diagrams below illustrate what happens when we initialise the variable `x` to 5 and then assign a new value to it.

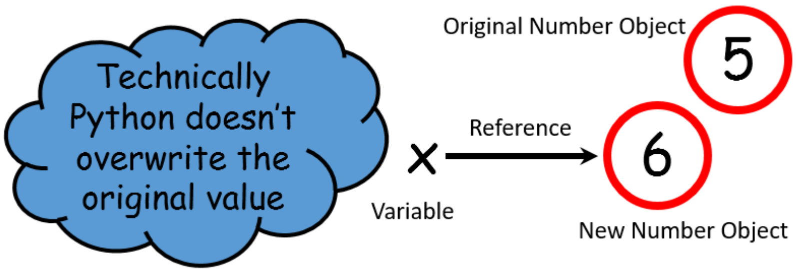


Memory representation following the statement `x = 5`



Conceptual, memory representation following the statement `x = x+1`

Conceptually, the value of a variable changes. The old value seems to be replaced with a new value. Technically, what happens is illustrated as follows:



The old value remains on in memory, and the variable that referenced it before the assignment now references the new value.

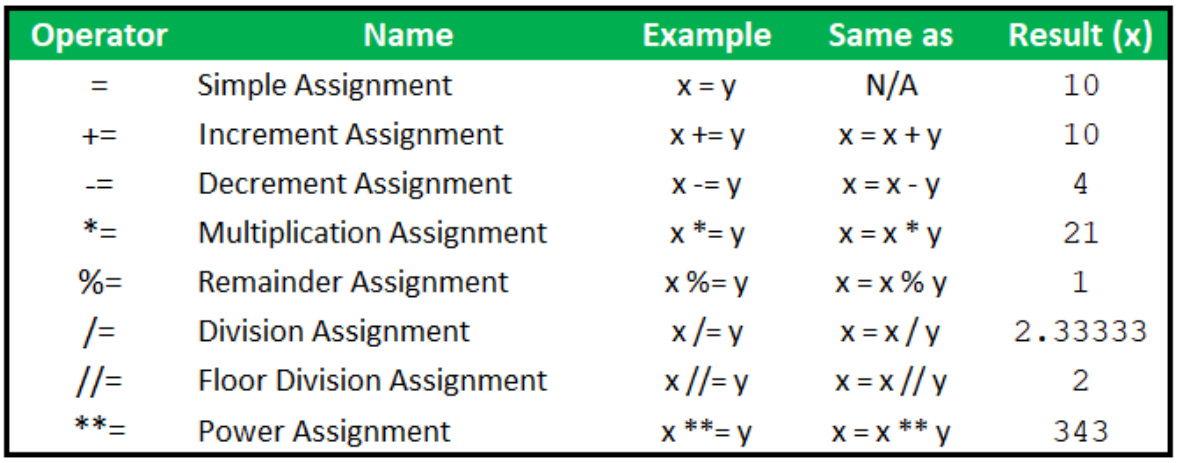
It is for this reason that objects such as strings and numbers are said to be *immutable*.

Finally, note that the original number object i.e. 5 in this case is left without a reference. Unreferenced values are referred to as *dangling objects*. Such objects are automatically returned by Python, to memory, in a process known as *garbage collection*.)

2. Python Assignment Operators

In addition to the simple assignment operator introduced and used throughout this section, Python supports a variety of other more compact assignment operators.

These are given here for the sake of completeness. (Assume **`x=7`** and **`y=3`**)



Operator	Name	Example	Same as	Result (x)
=	Simple Assignment	x = y	N/A	10
+=	Increment Assignment	x += y	x = x + y	10
-=	Decrement Assignment	x -= y	x = x - y	4
*=	Multiplication Assignment	x *= y	x = x * y	21
%=	Remainder Assignment	x %= y	x = x % y	1
/=	Division Assignment	x /= y	x = x / y	2.33333
//=	Floor Division Assignment	x //= y	x = x // y	2
**=	Power Assignment	x **= y	x = x ** y	343

Python Assignment Operators

Unlike C++, Java and some other programming languages, Python does not provide built-in support for the unary increment and decrement operators (e.g. $--x$ and $y++$).

STUDENT TIP
It is useful for teachers to be aware that novice programmers commonly confuse the use of the `print` command with an assignment statement.