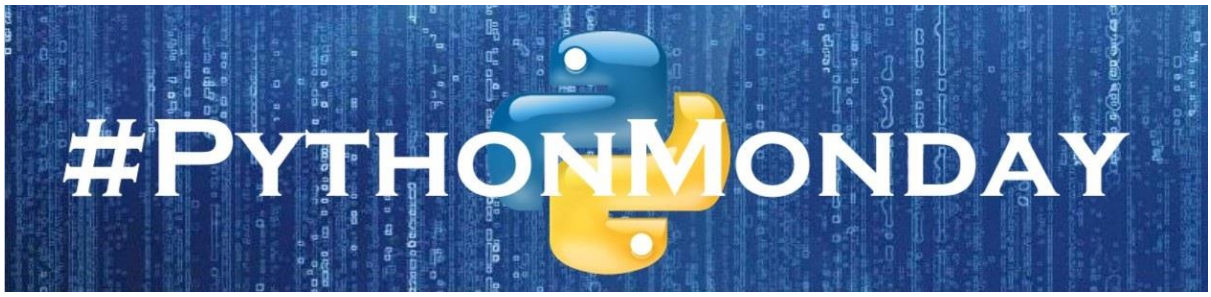

#PYTHONMONDAY

## *1. INTRODUCTION TO PYTHON*



## *2. VARIABLES AND VALUES*



## *3. PYTHON VERSIONS AND EDITORS*



## *4. THE "IF" STATEMENT*

## *8. FUNCTIONS AND METHODS*



## *9. TESTING PROGRAMS*



## *10. ARRAYS AND THE "FOR" STATEMENT*

# Introduction to #PythonMonday

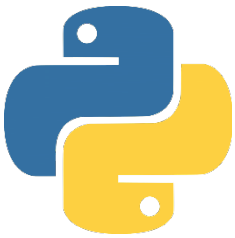
**Introduction**
I think it's really important that everyone has some idea as to how computer programming works, because a lot of decisions are being made all over the world by computer programs, and it's important to know how they are doing that.

It's important to learn to program, because:

- Programming effects communities
- Programming effects medicine
- Programming effects justice
- Programming effects business
- Programming effects governments
- Programming is power

I personally think programming should be taught to everyone, it is a very useful skill, and more importantly understanding programming helps you evaluate so many news stories about phone hacking scandals and about cloud leaks of private photographs; about facial recognition systems tracking people and about SmartSpeakers listening to conversations; and about bots and trolls on social media.

So we are going to learn how to program, and the first thing we need to decide on when learning to program is which programming language to use. We are going to use the Python programming language because it's an easy language to learn, and it's also a very powerful programming language. Here's the python logo:



So this is the plan, we'll learn to program, one page at a time, and every Monday a new page will be posted, that will be the beginning of a programming journey that we'll be taking together.

#PythonMonday © Damian Gordon

# What is Programming and Python?

**Programming Languages**

This bit is easy, a programming language is how we tell the computer what we want it to do. The problem is that computers don't really understand anything except ones and zeros (binary), but humans aren't that good at using binary, so we invented these programming languages to communicate with the computer because they are close to being a version of (very structured) English, but at the same time they are easy to translate into binary for the computer, so that everyone is happy. So the program that translates our work into binary (that the computer can understand) is called an "interpreter" (FYI, a different type of translator is called a "complier").

**Python**

Python is a programming language, designed by Guido van Rossum, who developed the first Python interpreter in 1991. He designed it to be easy-to-use and easy to write programs with. The name of the language is a tribute to the comedy group Monty Python, and their brand of zany humour is infused in a lot of Python reference materials.

The Tab key on your keyboard is really important when you are programming in Python. The spacing (or indentation) you put in your programs helps tell the interpreter where different sections of a Python program are, e.g.:

```
        Section 1
                Section 2
                        Section 3
                Section 4
        Section 5
```

But we'll talk more about that later.

Python has a core set of functions built into it, and there all loads of extra features you can add into it from known and trusted sources (called "libraries") that allow you to do even more fun stuff with Python. Some things Python is really good at include; storing data, interacting with web sites, creating nice visual content, controlling robots and other devices, designing games, etc.

**Open Source Software**

One last thing, many versions of the Pyton interpreter are free, and not copyrighted, so people can use them, modify them, improve them, and distribute them. This type of software is called "open source" software. The idea of which is that software should be free, and it goes back to the very roots of computer programming, where it was taken for granted that everyone would share programs so that programmers could learn from each other. Many people view this type of software as being important in a political sense, and important for democracy.

#PythonMonday © Damian Gordon

# Our First Python Program

**Finding a Python Interpreter**

We have two choices when it comes to finding a Python Interpreter (remember, this is the thing that changes our programs into something that the computer can understand); we can either use an online interpreter, or we can download one. The benefit of using the online option is that we don't need to install anything, and the benefit of downloading one is that, it will work even if you lose internet access.

For the moment, I'd recommend using an online interpreter, and we'll look at installing one on your computer later, some free online interpreters include:

| |
|---|
| https://www.python.org/shell/ |
| https://www.programiz.com/python-programming/online-compiler/ |
| https://repl.it/languages/python3 |
| https://www.onlinegdb.com/online_python_interpreter |

So pick any of the above webpages, and you should have a window to type commands in, and when you are finished, you can use the "Run" button (>) to run the program (NOTE: This is also called "executing the program"), and what that means is that the interpreter will change your program into binary for the computer.

**Our First Program: Hello, World!**

Our first Python program prints out a message to the screen, all we have to do is type in the following command:

```
print("Hello, World!")
```

And then run the program (by clicking the "Run" button), and if we have typed the program in correctly, we will see the following written on the screen:

```
Hello, World!
```

If we haven't typed it in exactly right, when we try to run the program, it won't work. Remember computers are really stupid, so it can't guess what you mean, you have to state your programs exactly correctly, and it will understand what you mean. If you leave out an inverted comma or bracket, the computer won't know what you mean, and will tell you there is an error, but don't worry, most programs will give you errors the first time you run them, so all you do is look for the error, fix it, and re-run the program.

Most programming courses and books (in every programming language) start with the "Hello, World!" program. It announces to the world that you are now a programmer, because once you complete your first program, you *are* a programmer. Also each program you write sees you contributing to the virtual world (cyberspace), so this first program means that you have started to claim a realm within cyberspace for yourself, and as you write more programs, you will find that your programs become more imbued with your spirit and persona.

#PythonMonday © Damian Gordon

# More on the Print Command

**The Point of Print**

The `print` command is probably one the most important commands, as it allows the computer to send messages to your screen, so everything from "`Hello, World!`" to all of your emails, webpages, and documents (including this file) are all created using the `print` command. Some version of this command is used to display every piece of text you see on a computer screen. Here's some terminology:

| | |
|---|---|
| **`Hello, World!`** | This is the message to be printed. |
| **`"Hello, World!"`** | Any message enclosed in inverted commas is called a *String* (or a *String of Characters*). |
| **`print("Hello, World!")`** | The `print` command takes in a String and puts it on the screen. (A fancy way of saying this is "The `print` function takes in a String as a parameter and puts it on the screen.") |

And as we saw before, when we run the `print` command above we get:

```
Hello, World!
```

If you want to leave a blank line after the message, Python has a special character sequence called the *newline* character (\n) to make that happen. All you have to do is add this into the String. So we can add it in at the end of the String, as follows:

```
print("Hello, World!\n")
```

And we will see the following written on the screen:

```
Hello, World!

```

Alternatively we can add the *newline* character to the start of the String:

```
print("\nHello, World!")
```

And we will see the following written on the screen:

```

Hello, World!
```

If we put the *newline* character in the middle of the String:

```
print("Hello,\nWorld!")
```

And we will see the following written on the screen:

```
Hello,
World!
```

So remember that we can put the *newline* character anywhere in the String, but it is also worth noting that we can get the same output without using the *newline* character by doing the following:

```
print("Hello,")
print("World!")
```

And this is a crucial point to remember; with most computer programs you write, there are multiple ways to achieve the same output, and it's up to you as the programmer to choose which approach you take. You have the power!

#PythonMonday © Damian Gordon

# Even More on the Print Command

**Using the Plus sign (+) with the Print Command**

If we want to join two strings together, we can use the plus sign (+) to do it:

```
print("Hello," + " World!")
```

And we will see the following:

```
Hello, World!
```

Please note that any time we put double quotes (") around something, it becomes a String, so even if we put double quotes around a number, it becomes a String, so to show that in action, try the following:

```
print("19" + " 91")
```

And we will see this output:

```
1991
```

So as we can see, because the numbers are enclosed in double quotes, the two Strings are joined together, it doesn't just add them up, it prints out the String "1991", which is the year Python was first created.

**Using the Multiply sign (*) with the Print Command**

As well as the plus sign (+), we can also use the multiply sign (*) to manipulate Strings. In Python, the multiply sign is the star character "*" (above the number 8 on your keyboard), unlike in maths where it's the X symbol (unfortunately the "X" looks too much like a capital "x", and so would be confusing, so the star is used instead). If we want to print the same message a number of times, we can use the multiply sign:

```
print("Hello, World!" * 3)
```

And we will see the following written on the screen:

```
Hello, World!Hello, World!Hello, World!
```

We can add a space in at the end of the Sting:

```
print("Hello, World! " * 3)
```

And we get:

```
Hello, World! Hello, World! Hello, World!
```

If we want to write each copy of the message on a new line, all we have to do is add the *newline* character to the end of the String:

```
print("Hello, World!\n" * 3)
```

And we will see the following written on the screen:

```
Hello, World!
Hello, World!
Hello, World!
```

If we want to use the multiply sign with Strings that have numbers in them:

```
print("9" * 3)
```

We will see the following written on the screen:

```
999
```

The year 999 was the date of a very famous Irish battle, the Battle of Glenmama.

#PythonMonday © Damian Gordon

# Doing Maths in Python

**Addition using the Plus sign (+)**

If we want to see how maths works in Python, let's us the print command to see what happens. So as we saw previously, the plus sign (+) can be used to join together two strings, but it can also be used to add up two numbers, as follows:

```
print(19 + 91)
```

And we will see this output:

```
110
```

So if the numbers don't have double quotes around them, Python treats them as numbers, and adds them up.

**Subtraction using the Minus sign (-)**

The minus sign (-) can be used to subtract two numbers, as follows:

```
print(19 - 91)
```

And we will see this output:

```
-72
```

So again, because the numbers don't have double quotes around them, Python treats them as numbers, and subtracts them.

**Multiplication using the Multiply sign (*)**

As we mentioned previously, the multiply sign in Python (and many other programming languages) is the star sign (*), so as long as the values are two numbers, they will be multiplied as follows:

```
print(19 * 91)
```

And we will see this output:

```
1729
```

So again, Python treats them as numbers, and multiples them.

**Davison using the Divide sign (/)**

The divide sign in Python (and many other programming languages) is the forward slash (/), so as long as the values are two numbers, they will be divide, for example:

```
print(19 / 91)
```

And we will see this output:

```
0.2087912087912088
```

So as we can see, Python divides the two numbers (both of which are whole numbers), and the result is a number with decimal places. In Maths (and in computers) we have a special name for any number that has a decimal place; we call it a *Real Number*. So for example 0.20879 is a Real Number, so is 3.14159, and so is 93.0 - as long as it has a decimal place, it a Real Number, if the number doesn't have a decimal place, we call it a *Natural Number*. So, for example, 93, -34, 2 are all Natural Numbers.

#PythonMonday © Damian Gordon

# Other Types of Division

**Regular Division (/)**

As we've seen already, the divide sign is the forward slash (/), and as long as the values are two numbers, they will be divide, for example:

```
print(11 / 4)
```

And we will get the following output:

```
2.75
```

But what if we want to just find out how many times the bottom number (denominator) fully divides into the top number (numerator). In this case it goes two (2) times with a remainder of three (3).

**Integer Division (//)**

So if you just want to find out how many times the denominator divides evenly into the numerator, we use two the forward slashes (//), so for example:

```
print(11 // 4)
```

And we will see this output:

```
2
```

Here's a few more examples:

| | | | | | |
|---|---|---|---|---|---|
| `print(11 // 1)` | `11` | `print(11 // 2)` | `5` | `print(11 // 3)` | `3` |
| `print(11 // 4)` | `2` | `print(11 // 5)` | `2` | `print(11 // 6)` | `1` |

**Division Remainder (%)**

And if you just want to find out what's the remainder when you divide the denominator into the numerator, we use the percentage sign (%), so for example:

```
print(11 % 4)
```

And we will see this output:

```
3
```

Because there three (3) left over when you divide four (4) into eleven (11) twice.

Here's a few more examples:

| | | | | | |
|---|---|---|---|---|---|
| `print(11 % 1)` | `0` | `print(11 % 2)` | `1` | `print(11 % 3)` | `2` |
| `print(11 % 4)` | `3` | `print(11 % 5)` | `1` | `print(11 % 6)` | `5` |

So as we can see, one (1) divides evenly into eleven (11), there's no remainder. When we divide eleven (11) by two (2), it goes five times and there's a remainder of one (1), when we divide eleven (11) by three (3), it goes three times and there's a remainder of two (2), when we divide eleven (11) by four (4), it goes two times and there's a remainder of three (3), etc.

These types of division might seem fairly trivial, but they are really powerful, and they allow you to write all kinds of programs, like checking if a number is even or odd; displaying the time in seconds, minutes, and hours; or doing some activity every second time, every third time or every Nth time.

#PythonMonday © Damian Gordon

# Working with Variables

**What is a variable?**
One of the most important things a computer program has to do is remember things, so for example, the bank has a program that remembers your bank balance, the email system has a program that remembers your password, the web browser has to have a way of remembering your favourite webpages. To help us understand how these programs remember things, we'll talk about variables.

We already know the idea of a variable from maths, I'm sure we can all remember having to solve equations like the following:

$$2X - 10 = 0$$

And we would do the following:

$$2X = 10$$
$$X = 5$$

In another problem we might get the following:

$$3X + 12 = 0$$

And we would do:

$$3X = -12$$
$$X = -4$$

So the general idea is clear, the variable "X" is used to represent a number, and at different times the variable can represent different values, the same way the value stored in your bank account can change over time ;-)

**Variables in Programming**
In programming, variables work in a similar way, but instead of working out what value a variable has, you as the programmer have to tell the computer the value of the variable. And at another point in the program, you can change that value.
In Python we can tell the computer the value of the variable as follows:

```
X = 5
```

So what we are telling the computer is that "the variable X is equal to 5", or a better way of saying it is "X gets the value 5" or "X is assigned the value 5".

***Damian's Concept of Variables***

I like to think of a variable as a metal bucket, and the name of the variable is painted on the outside of the bucket, and you can put a number into the bucket, and later on you can take that number out of the bucket and replace it with another number.


#PythonMonday © Damian Gordon

# Printing Out Variables

**Why do we need to print?**

We can assign the variable "X" the value 5 as follows:

```
X = 5
```

And if we do this command, what will be output onto the screen?

```

```

Nothing is the answer, there's no instruction to print anything onto the screen, so to see the value of "X" we need to do the following:

```
print(X)
```

And we will get the following on the screen:

```
5
```

So if we want to find out the value of a variable, one way of doing it is to print it to the screen, we just put the name of the variable inside the print brackets, and most importantly we have to remember that we should not put the variable name in inverted commas (""), because if we do, this happens:

```
print("X")
```

And we will get the following on the screen:

```
X
```

So without the inverted commas we get the value of the variable, but with the inverted commas, it just prints out the name.

**Changing the Output**

We could also try the following:

```
print(X + 1)
```

And we will get the following on the screen:

```
6
```

We should note that the value of "X" hasn't changed, it's still 5, but you have printed out that value plus one. So if we follow that command with this one:

```
print(X)
```

We will still get the following:

```
5
```

So "X" is still 5, and if we follow that command with this one:

```
print(X + 10)
```

We will get the following:

```
15
```

So the important thing to remember is that we can use the print command to see the value of a variable, but the print command can't change the value of "X", it can just print the value out, or print out some calculation based on it.

Understanding how variables work is very important, they are used in programs to store all kinds of data and it's important to know how to use them, and how to display the values they store.

#PythonMonday © Damian Gordon

# More with Variables

**More than one Variable**

Let assign the variable "X" the value 5:

```
X = 5
```

And in Python every time we want to create a new variable all we have to do is assign it to a value, so then let's assign the variable "Y" the value 15:

```
Y = 15
```

So if we say:

```
print(X)
```

we will get the following on the screen:

```
5
```

And if we say:

```
print(Y)
```

We will get the following on the screen:

```
15
```

**Variables working together**

We could do the same thing in a slightly different way, if "X" is assigned 5:

```
X = 5
```

And then the next command we give is the following:

```
Y = X + 10
```

What value do you think "Y" has? Well we can find out with the following command:

```
print(Y)
```

we get the following on the screen:

```
15
```

So we can assign a variable its value based on the value of another variable, but it's important to be aware that if the first variable ("X") later changes its value, that won't impact the value of the second variable ("Y"), unless we repeat this initial assignment statement again:

```
Y = X + 10
```

**Printing Out Variables**

If we want to print out the sum of the two variables, all we have to do is:

```
print(X + Y)
```

And we get the following on the screen:

```
20
```

If we want to see the values of the individual variables, we list all the variables we want printed out, separated by a comma:

```
print(X, Y)
```

And we get the each value on the screen separated by a space:

```
5  15
```

#PythonMonday © Damian Gordon

# Types of Variables

**Numeric Variables**

We have already seen that we can create variables for integers (whole numbers that are either positive or negative):

```
X = -5
```

And we can create variables for real numbers (numbers with a decimal place), and these are sometimes called *floats* (or *floating point numbers*):

```
Y = 15.45
```

So the number is a real if it has a decimal place, even if it is followed by a zero:

```
Z = 14.0
```

So we can see that if a computer program needs to remember your bank balance, the speed of your car, or the price of your shopping; a numeric variable is a good way to store those values. But there are other things that computer programs need to remember, like your name and address, your list of favourite websites, or the password for your email; that can't be stored as numbers.

**Alphanumeric Variables**

So how do we create variables that can store a single character, or a string of characters? Well the good news is that it's exactly the same way as numeric values, so for example, to assign "X" to the string "Hello, World!", we do it as follows:

```
X = "Hello, World!"
```

And we can see what value "X" has by doing the following:

```
print(X)
```

And we will get the following on the screen:

```
Hello, World!
```

Similarly, to assign "Y" to the character 'B', we do it as follows:

```
Y = 'B'
```

And we can see what value "Y" has by doing the following:

```
print(Y)
```

And we will get the following on the screen:

```
B
```

Note, as a convention, we use a single quote (') for single characters and double quotes (") for a string of characters. Also remember that we can have a string that is made up of numbers, for example:

```
X = "1991"
```

And this is not the same as if we do:

```
Y = 1991
```

In the first case we have a string that we can print out and add to other strings, and in the second we have a number that we can add, subtract, multiply and divide. So we can do something like the following in the first case, but not the second:

```
print(X + " is the year Python was created")
```

#PythonMonday © Damian Gordon

# Boolean Variables

**A Special Type of Variable**

There is one other type of variable that is worth mentioning, and that is a variable that will store only one of two values, *True* or *False*. The type of variable is called Boolean, and is named after mathematician George Boole. So to create a variable of this type we do it the same way as any other variable assignment:

```
X = False
```

And we can see what value "X" has by doing the following:

```
print(X)
```

And we will get the following on the screen:

```
False
```

**The NOT Function**

So if we want to print out the opposite of what is stored in a particular variable, we can use the *not* function. So if the variable "X" has a value of *True*, as follows:

```
X = True
```

And if we do the following:

```
Y = not(X)
```

Then if we print "Y":

```
print(Y)
```

We will get:

```
False
```

So the *not* function tells you the opposite, so if the variable "X" represents if someone is over 18 years old, then *not*(X) represents if someone is not over 18.

We can also do the following:

```
Z = not(not(X))
```

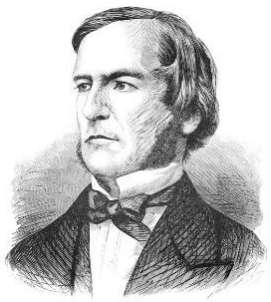
Then if we print "Z":

```
print(Z)
```

We will get:

```
True
```

**BIOGRAPHY: George Boole**



Boole was born in Lincoln on 2nd November 1815 and died in Cork on 8th December 1864. He was a self-taught mathematician who was the first professor of mathematics at Queen's College, Cork (now called *University College Cork*). His most important work on symbolic logic that was contained in his monograph "*The Laws of Thought*", and focuses on the use of *True* and *False* to help automate different decision-making processes, which is used in a wide range of fields, including programming & designing circuits.

#PythonMonday © Damian Gordon

# Names of Variables

**What Can We Call a Variable?**

So far we have only used a since letter to represent a variable, and that is because that is the most common way they are used in mathematics, so for example if we wanted to represent *pi*, the ratio of a circle's circumference to its diameter:

```
P = 3.1415926536
```

But in computer programming we can call a variable name almost anything we want, and usually we try to create variable names that are as descriptive as possible, this is important so that other people can read our programs and easily understand what they do, particularly since a lot of programs we write will be done as part of a team:

```
Pi = 3.1415926536
```

Calling this variable `Pi` seems like it might be sufficient when we create the variable, but we might later realise that we need to do some calculations with *pi* just as 3.14, and other calculations with *pi* with ten decimal places, so we can do, the following:

```
TheValueOfPiToTwoDecimalPlaces = 3.14
```

And we also have:

```
TheValueOfPiToTenDecimalPlaces = 3.1415926536
```

**What Can't We Call a Variable?**

There are only a limited number of specific words that cannot be used as variable names, for example, we know that the words `True` and `False` have a specific meaning in Python, and so we shouldn't use them as variable names as they are already reserved for a different use (they are called *reserved words*). There are also several built-in functions and features in Python and we shouldn't use their names as variable names, for example, we know that the function `print` tells the system to write messages to the screen, so we can't use that as a variable name. We'll present a full list of names we can't use as variables on the next page.

**Camel Case**

Capitalizing each word in a variable name makes it easier to read, for example, `ThisIsAVariable`. There are alternatives, but they have their own drawbacks:

- Spaces: `This is a variable`
  Using spaces to separate the words in a variable name causes some systems to treat them like separate variables.
- Dashes: `This-is-a-variable`
  Using dashes to separate the words in a variable name causes some systems to treat the dashes as if they are the subtract symbol.
- Underscores: `This_is_a_variable`
  Using underscores to separate the words in a variable name can sometimes look like spaces particularly if the editor you are using to type in the commands changes the colour of certain words or adds underlines.

#PythonMonday © Damian Gordon

# Python Keywords and More

**Python Keywords**

There are specific words that have a pre-existing meaning to the Python interpreter, these are referred to as *keywords* or *reserved words*, and cannot be used as variable names. We have seen that `True` and `False` are reserved, and in total there are 35 keywords in the current version of Python (which we'll learn a lot more about in subsequent pages), as follows:

| | | | | |
|---|---|---|---|---|
| **and** | **continue** | **finally** | **is** | **raise** |
| **as** | **def** | **for** | **lambda** | **return** |
| **assert** | **del** | **from** | **None** | **True** |
| **async** | **elif** | **global** | **nonlocal** | **try** |
| **await** | **else** | **if** | **not** | **while** |
| **break** | **except** | **import** | **or** | **with** |
| **class** | **False** | **in** | **pass** | **yield** |

**Python Built-In Functions**

Other words that we have to avoid using when we are creating variables are the names of built-in functions. The `print` command is an example of a built-in function, so when we do `print("Hello, World!")`, we call the function name – `print` – we follow it with brackets, that encloses some content, in this case it's the string `"Hello, World!"` but we've seen it could also be a number, a character, or even a Boolean. And we know that a built-in function does something, so in this case, it writes the string you have enclosed in brackets onto the screen. All functions work the same way, they take in value and perform a particular operation:

| | | | | | |
|---|---|---|---|---|---|
| **abs** | **compile** | **format** | **isinstance** | **object** | **set** |
| **all** | **complex** | **frozenset** | **issubclass** | **oct** | **setattr** |
| **any** | **delattr** | **getattr** | **iter** | **open** | **slice** |
| **ascii** | **dict** | **globals** | **len** | **ord** | **sorted** |
| **bin** | **dir** | **hasattr** | **list** | **pow** | **staticmethod** |
| **bool** | **divmod** | **hash** | **locals** | **print** | **str** |
| **bytearray** | **enumerate** | **help** | **map** | **property** | **sum** |
| **bytes** | **eval** | **hex** | **max** | **range** | **super** |
| **callable** | **exec** | **id** | **memoryview** | **repr** | **tuple** |
| **chr** | **filter** | **input** | **min** | **reversed** | **type** |
| **classmethod** | **float** | **int** | **next** | **round** | **vars** |

#PythonMonday © Damian Gordon

# Versions of Python

| Version | Python v1 | Python v2 | Python v3 |
|---|---|---|---|
| Years | 1991-2000 | 2000-2010 | 2008-To Date |

**Python Version 1**
Python was created by Dutch programmer Guido van Rossum, and the first interpreter was released in 1991. This initial version had almost everything van Rossum wanted in the language, but there were a few small things he felt were missing, so instead of labelling this as version 1.0, he released it as 0.9. It wasn't until 1994 that version 1.0 of the interpreter was finally released and had new features to help process lists and records. Python became more and more popular, and programmers wanted more features added to the interpreter, both to make it better and to give it all the features that other programming languages have. For the next six years new features were added to the interpreter, and a new sub-version of the interpreter was released on average once a year, until reaching version 1.6.

**Python Version 2**
At this point van Rossum decided to look at all the new features that had been added on, as well as looking at all the major outstanding issues, and decided to do a major redesign of the interpreter, and release an updated interpreter as version 2.0 in the year 2000. This new version added further features to help process lists and records, as well as changing the way that Python stores information in computer memory. New sub-versions of Python were created again on average once a year until 2008, when van Rossum decided to redesign the interpreter again, and release a new version. However, unlike the previous transition from version 1 to version 2, the new redesign involved a complete change of how the interpreter worked, to such an extent that a Python program written in version 2 would not work with a version 3 interpreter. This caused a fork in the road, where some people in the Python community continued to develop programs in version 2 and continued to develop and improve the version 2 interpreter for the next two years, and others began to develop programs in the new version 3 interpreter. The last new improvement of version 2 interpreter was released in 2010 and was version 2.7, which some programmers in the Python community still use.

**Python Version 3**
Version 3 of Python was a complete redesign that began in 2008, it looked at all of the features that had been added since 1991, to look at which ones were duplicated and which ones were consistent with the overall Python philosophy of "*there should be one - and preferably only one - obvious way to do it*", and tried to remove all of the unnecessary features. There have been 9 sub-versions so far, with Python 3.10 anticipated to come out on the 25th of October 2021.

#PythonMonday © Damian Gordon

# Full Versions Table

**Full List of Versions (and Sub-Versions) of the Python Interpreter**
This is a complete list of all the versions of the Python interpreter since 1991, including their release date and the end date of the full support by the Python community.

| Version Number | Last Micro-Version | Release Date | End of Full Support |
|---|---|---|---|
| **0.9** | 0.9.9 | 20th Feb 1991 | 29th Jul 1993 |
| **1** | 1.0.4 | 26th Jan 1994 | 15th Feb 1994 |
| **1.1** | 1.1.1 | 11th Oct 1994 | 10th Nov 1994 |
| **1.2** | 1.2.0 | 13th Apr 1995 | No support |
| **1.3** | 1.3.0 | 13th Oct 1995 | No support |
| **1.4** | 1.4.0 | 25th Oct 1996 | No support |
| **1.5** | 1.5.2 | 3rd Jan 1998 | 13th Apr 1999 |
| **1.6** | 1.6.1 | 5th Sep 2000 | 30th Sep 2000 |
| **2** | 2.0.1 | 16th Oct 2000 | 22nd Jun 2001 |
| **2.1** | 2.1.3 | 15th Apr 2001 | 9th Apr 2002 |
| **2.2** | 2.2.3 | 21st Dec 2001 | 30th May 2003 |
| **2.3** | 2.3.7 | 29th Jun 2003 | 11th Mar 2008 |
| **2.4** | 2.4.6 | 30th Nov 2004 | 19th Dec 2008 |
| **2.5** | 2.5.6 | 19th Sep 2006 | 26th May 2011 |
| **2.6** | 2.6.9 | 1st Oct 2008 | 24th Aug 2010 |
| **2.7** | 2.7.18 | 3rd Jul 2010 | 1st Jan 2020 |
| **3** | 3.0.1 | 3rd Dec 2008 | 13th Feb 2009 |
| **3.1** | 3.1.5 | 27th Jun 2009 | 12th Jun 2011 |
| **3.2** | 3.2.6 | 20th Feb 2011 | 13th May 2013 |
| **3.3** | 3.3.7 | 29th Sep 2012 | 8th Mar 2014 |
| **3.4** | 3.4.10 | 16th Mar 2014 | 9th Aug 2017 |
| **3.5** | 3.5.9 | 13th Sep 2015 | 8th Aug 2017 |
| **3.6** | 3.6.11 | 23rd Dec 2016 | 24th Dec 2018 |
| **3.7** | 3.7.8 | 27th Jun 2018 | 27th Jun 2020 |
| **3.8** | 3.8.5 | 14th Oct 2019 | 30th Apr 2021 |
| **3.9** | 3.9.0 | 5th Oct 2020 | 31st May 2022 |
| **3.10** | 3.10.0 | 25th Oct 2021 | 31st May 2023 |

#PythonMonday © Damian Gordon

# Backward and Forward Compatibility

**Introduction to Compatibility**
Have you ever bought a new phone and found out that the charger for your old phone works with the new phone? Have you ever found the opposite, where you buy a different phone and you find that an old charger doesn't work with the new phone (even if the phone is from the same manufacturer)? This is an issue we call "compatibility", so if the manufacturer keeps the same charging port (socket) in different models of the phone, then the charger will work on all of them, but if they change the shape of the port (or the power requirements of the port), then the charger isn't compatible with the newer version.

**Backward Compatibility**
*Backward Compatibility* means that the current version of some technology works with older versions of that technology. Backward compatibility is also sometimes called *Downward Compatibility*. If an organisation changes a technology so that it is no longer compatible, they are said to be "breaking" backward compatibility. If we have a phone that we have downloaded apps onto and bought chargers and other technologies for it; if we get an upgrade to that phone and none of the apps work on it, and none of our technologies work on it, we would be much more likely to move onto a new brand of phone than if some (or all) of the existing technologies work with the new phone, so it's better for tech companies to ensure their technologies are backwards compatible. So, for example, when Sony released the PS2, it was backwards compatible with the PS1 and all of the PS1 games, and so anyone who bought the PS2 already had a load of games available for them to play.

The big challenge with backward compatibility is that it requires each new addition and innovation to any given technology to work in such a way that it doesn't counteract, or in any way impact, the existing features of the technology. This can sometimes lead to higher costs in developing systems, and it can curtail innovations.

**Forward Compatibility**
*Forward Compatibility* means that the current version of some technology is designed in such a way that it will work with future versions of that technology. Forward compatibility is also sometimes called *Upward Compatibility*. To make the system forward compatible, it doesn't mean that the designer has to predict each new future innovation and deal with it, instead if the older system is able to take in whatever inputs are necessary for that system, and ignore any other inputs (that may be used by newer versions), it will be compatible. In this scenario, the system may not be fully backward compatible, but is forward compatible.

Sometimes tech companies will deliberately ensure that a technology isn't forward compatible so that it will force all of their customers to purchase new versions of the technology and all of the related technologies. This can be seen as a way to drive sales.

**Python**
Python 2 is backwards compatible with Python 1, and Python 1 is forward compatible with Python 2. Python 3 is not backwards compatible with Python 2, and Python 2 is not forward compatible with Python 3.

#PythonMonday © Damian Gordon

# Python Editors

**Introduction to Python Editors**
So far we have written programs that can be written one line at a time and typed directly into the Python window (Shell), so to write programs with multiple lines we can also use programs called *IDEs*, or *Integrated Development Environments*, that work in much the same way as a text editor like Microsoft Word works – you open a screen and start typing text starting in the top left corner, and use the "ENTER" key to bring yourself onto the next line. Also, a lot of text editors will highlight words that appear to be misspelled or have grammar issues, and in the same way IDEs will highlight keywords (as detailed on Page 14), and issues with grammar (or more correctly *syntax*). There are many possible Python editors (or IDEs) available, and each have their benefits, but when you download the Python interpreter, it already comes with an editor called *IDLE*, which is a very good editor, so we'll start there.

**IDLE**
According to Python creator Guido van Rossum, IDLE stands for "Integrated Development and Learning Environment", but since we know van Rossum is a big fan of Monty Python, it's likely it's also a tribute to Eric Idle. The IDLE editor allows you to open multiple editing windows, it highlights keywords and syntax, it helps with the formatting of text, and has tools to help you find and fix errors in your programs (these are called *debugging tools*).

**PyCharm**
A popular alternative to IDLE is PyCharm which has all of the main features of IDLE, as well as the ability to navigate and manage larger programs more easily, and manage multiple versions of the same program easily. It also works well with tools for building web applications, and it's also very easy to change the PyCharm interface, you can choose different themes, colour schemes, and change what happens when you press the function keys on your keyboard.

**Spyder**
Spyder also has similar features to IDLE but focuses on looking at how well the programs you write are performing – so are they efficient? Or are there parts of the program that could be written more simply? It also allows you to edit the program code in more visual ways, and is very useful for people who are creating programs for data analysis and data science.

**Jupyter Notebook**
Jupyter Notebook is part of a larger project called *Project Jupyter* which is a web-based editor that helps you write programs in Python that work easily that work with web applications and other web-based program. It also has some nice visualisation tools that make it useful data analysis and data science programs (like Spyder).

**Atom**
Atom is an editor that can be used by a wide range of programming languages and provides all of the features of IDLE, and it also integrates with version control systems, and has a big community of programmers who keep adding new features to Atom, so you have to try to keep up with all the new features. It can also be integrated easily with a range of databases.

#PythonMonday © Damian Gordon

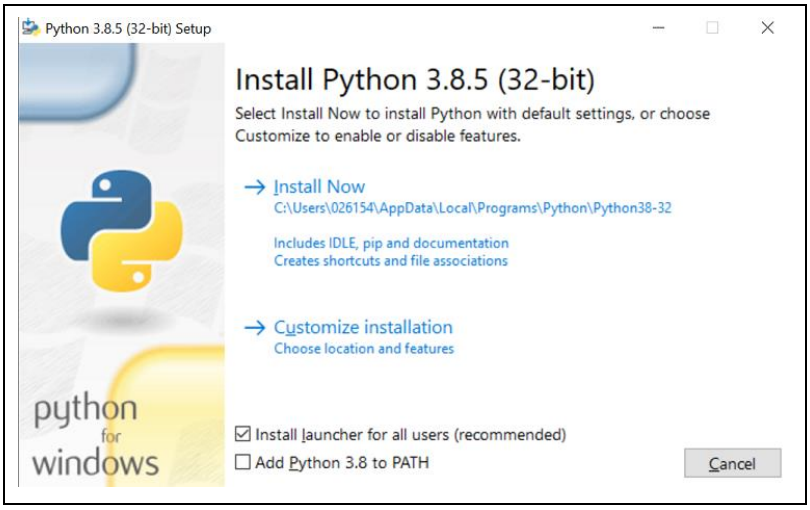
# Downloading Python

**The Python Site**

The main Python site is www.python.org, where you can download a wide variety of tools to help you when developing python programs, and the "Downloads" section is where you will find a variety of interpreters. The web address is: https://www.python.org/downloads/

The downloads page typically has the latest version of the Python interpreter at the top of the page, usually in blue, with a yellow button to download the latest version:


Once you click on the download button, an *.EXE* will start to download onto your computer, when it is completed, click on the executable, and it will run the installation process. You will get a window like the following, and select "Install Now":

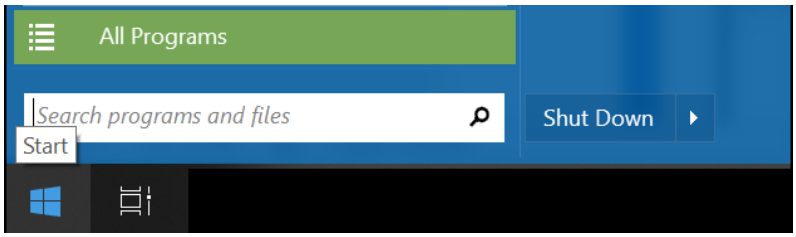



The installation process will take a few seconds, and then the Python interpreter, the IDLE editor, and some other tools will be installed on your computer.

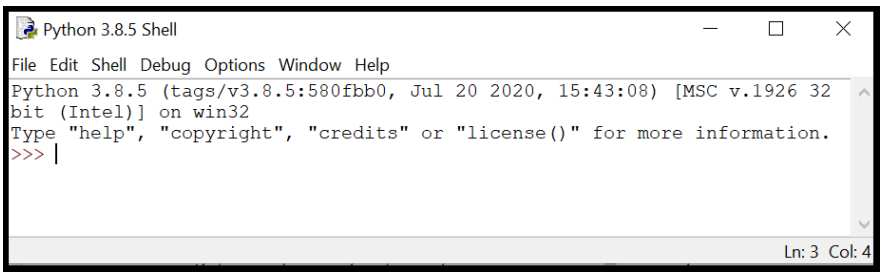
# Working with IDLE

**Getting Started with IDLE**

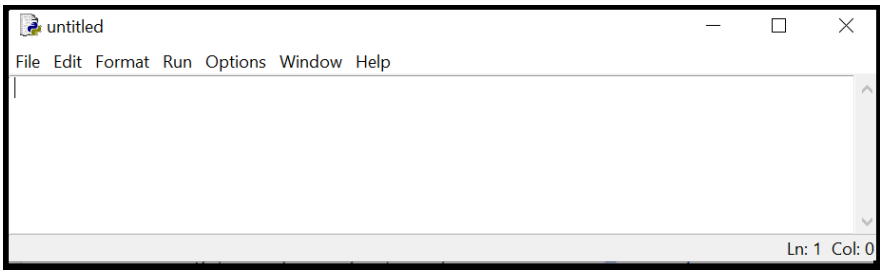
Once you have installed the Python interpreter, you can begin to write programs by clicking on the Windows "Start" button in the bottom left-hand side of the screen, and type in the word "idle" into the search box that appears.




You will be presented with a list, and under the heading of "Programs" you will see an option called "IDLE (Python)", click on it and a window like this will open:




This window is your **output window**, all of the results of your programs will be printed out in this window. The title of the window is always going to be "Python XX Shell". So the next thing we need to do is to create an input window, so if you click on the "File" menu, and select "New File", and you will get a new window like this:

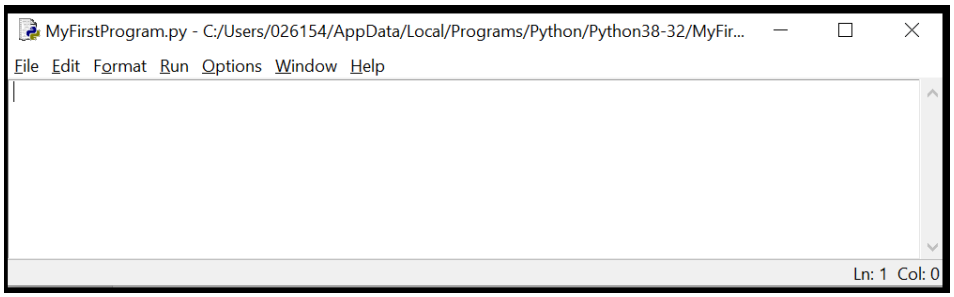



This window is your **input window** and the good news is that it works almost exactly the same as a notepad file, so if you select the "File" menu, and click on the "Save As…" option, you can give your program a name. So if, for example, you want to the program a particular name like "MyFirstProgram", and click on the "Save" button. All Python programs have an extension of ".py" Once you have written a few Python programs you can select the "File" menu, and click on the "Open" option and that will allow you to select a particular file you have already saved.

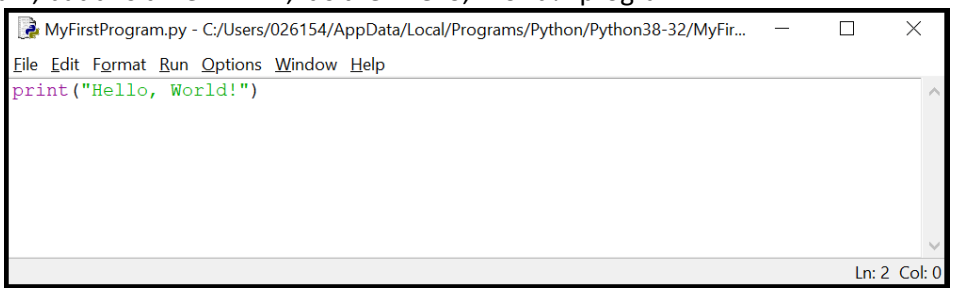
# Our First Program in IDLE

**MyFirstProgram.py**

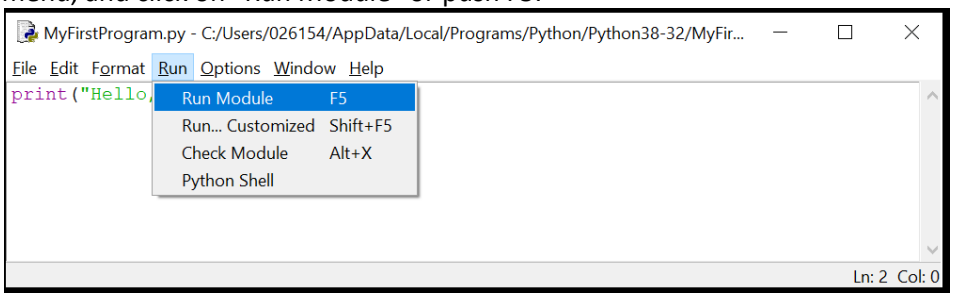
Once we have opened IDLE, and created a new file that we have saved, we should see the name of that file in the title bar of the window:



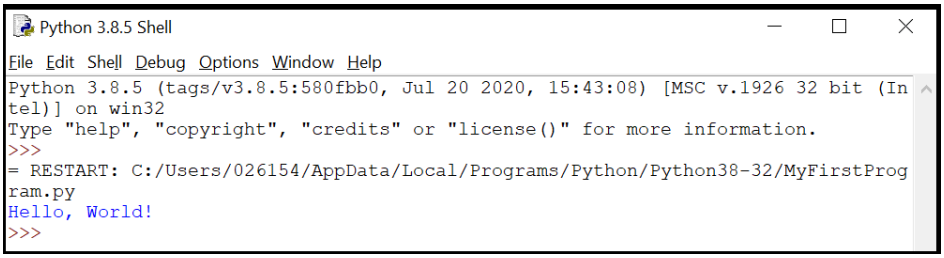

Now we can type in a program, so for our first program let's do the very first program we did again, but this time in IDLE, it's the "Hello, World!" program:




Now save it, by clicking on "File" and "Save", and to run the program go to the "Run" section of the menu, and click on "Run Module" or push F5.




Now the original Python Shell window will pop up again, and the output of our program will be written into that window:



# Using the IF Statement

**Adding Choice**

Almost every program we write will want offer the *end-user* (the person using the program) some kind of choice; examples of which include: "Do you wish to continue?", "Do you want to print a receipt?", "Do you want to do this transaction in Euro or Dollars?", "Do you agree to the Terms and Conditions?". It is really important to give the end-users a choice because we want to give them as much control as possible to allow our programs to work for them, instead of having them try to conform to the programs. As discussed before programming is a political act, and by giving the end-users a choice you are helping give them some control over the systems that can have a very significant impact on many aspects of their lives.

In Python, one way to give the end-users a choice is to use the IF statement, which has two possible paths, and you pick one path or the other, based on some condition. So, for example:

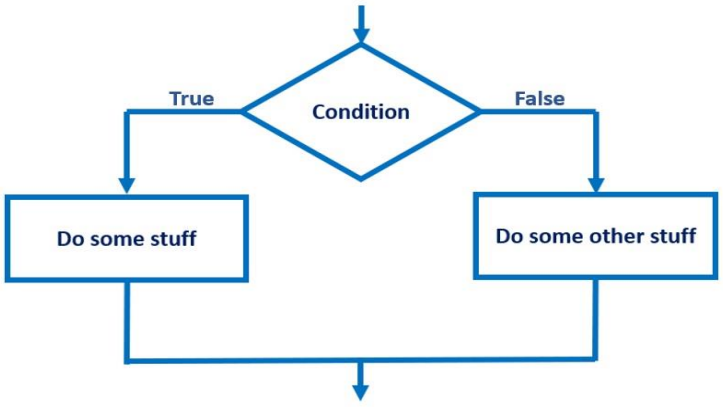
**if** (today is Saturday or Sunday)**:**
    then it's the weekend, so chill out,
**else:**
    it's a workday, so go to work or college.

***The IF Statement***

```
if (condition):
        then do some stuff
else:
        do some other stuff
```

Or as a picture, we can show the IF Statement like this:

***The IF Statement as a Flow Chart***




The diagram above is called a *Flow Chart*.

#PythonMonday © Damian Gordon

# Our First IF Statement

**The IF Statement**

***The IF Statement***

```
if (condition):
        do some stuff          [this is called the "consequent"]
else:
        do some other stuff    [this is called the "alternative"]
```

We've mentioned the "condition" already, and it is worth emphasising that the condition has to be something that is either "True" or "False" (a Boolean), so for example, is today Monday will either be true or false. If the condition evaluates to "True" then the *consequent* statements are run, if it evaluates to "False" then the *alternative* statements are run.

It is worth remembering that the "if" and the "else:" lines are at the same level of indentation, and that both the *consequent* and *alternative* statements are one tab (four spaces) in from those.

**Our First IF Program**

Below is our first program with an IF Statement. It works as follows, we create a variable called X, and set it to be the number 5, then we check if X is greater than 10, and if it is, we print out the *consequent* phrase after the condition, and if it isn't we print out the *alternative* phrase after the "else:".

***Sample IF Statement***

```
X = 5
if (X > 10):
        print("X is bigger than 10")
else:
        print("X is less than or equals 10")
```

We note that the condition is "`X > 10`", and it is either True or False, so either the variable in X is either great than 10 or it isn't. If it is greater than 10, it will print out the phrase "`X is bigger than 10`", and if it is less than or equals to 10, it will print out the phrase:

```
X is less than or equals 10
```

In this case X is less than 10 so it will print out the phrase after the "else:". To test this program write it into IDLE, run it, and check the result. Then change the first line of the program and replace with number 5 with the number 15 and run the program again, then try 10, then try 0, and try -10, to check this all makes sense.

Please note again the indentations, the IF statement is at the same level of indentation program code before that statement, as is the "else:", but the *consequent* and *alternative* print statements go in by one tab of indentation.

#PythonMonday © Damian Gordon

# More on the IF Statement

**Better Variable Naming**

When writing our programs, in practice, we will try to use sensible and descriptive variable names, so if we wanted to check if someone was born after 1967:

```
DateOfBirth = 1985

if (DateOfBirth > 1967):
    print("Date of birth is after 1967")        ← consequent
else:
    print("Date of birth is not after 1967")    ← alternative
```

And we will see this output:

```
Date of birth is after 1967
```

So if the DataOfBirth variable is great than 1967 the "consequent" is printed out, otherwise if the value is less than or equal to 1967 the "alternative" is printed out.

**Compound IF Statements**

It is also worth noting that it is possible to make the "condition" part of the IF Statement a compound one (made up of two parts or more) using the "and" statement. So for example, if we want to check if today is Monday and the month is October, the condition part of the IF Statement could look as follows:

```
if (Today == "Monday" and ThisMonth == "October"):
   print("It's a Monday in October")
else:
   print("It's not Monday and/or not October")
```

And only if both conditions are true, then the overall condition evaluated as true, if either is false (or both are false), then the overall statement is evaluated to be false. We note that to check if two things are equal in Python we use the "= =" statement.

Another example could be to check to see if someone was born after 1967 but before 2002, so we could write something as follows:

```
DateOfBirth = 1985

if (DateOfBirth > 1967 and DateOfBirth < 2002):
  print("Date of birth is in the range 1968-2001")
else:
  print("Date of birth is not in the range 1968-2001")
```

And we will see this output:

```
Date of birth is in the range 1968-2001
```

We note that to check if something is in the range of 1968-2001, our condition has to check if the variable is greater than 1967 and less than 2002. We note that the opposite of "greater than" is "less than, or equal to", and the opposite of "less than" is "greater than, or equal to".

#PythonMonday © Damian Gordon

# Conditional and Logical Operators

**Conditional Operators**

We've seen some of the conditional operators already, here is the full list:

| | | | |
|---|---|---|---|
| **= =** | *is equal to* | **! =** | *is not equal to* |
| **>** | *is greater than* | **<** | *is less than* |
| **> =** | *is greater than or equal to* | **< =** | *is less than or equal to* |

**Logical Operators**

We've seen some of the logical operators already, here is the full list:

**AND**

If you have two conditions, Cond1 and Cond2, then:

| ***Cond1*** | ***Cond2*** | ***Cond1 and Cond2*** |
|---|---|---|
| *False* | *False* | ***False*** |
| *False* | *True* | ***False*** |
| *True* | *False* | ***False*** |
| *True* | *True* | ***True*** |

The overall condition is True, if both individual conditions are True.

**OR**

If you have two conditions, Cond1 and Cond2, then:

| ***Cond1*** | ***Cond2*** | ***Cond1 or Cond2*** |
|---|---|---|
| *False* | *False* | ***False*** |
| *False* | *True* | ***True*** |
| *True* | *False* | ***True*** |
| *True* | *True* | ***True*** |

The overall condition is True, if either or both conditions are True.

**NOT**

If you have a condition, Cond1, then:

| ***Cond1*** | ***not(Cond1)*** |
|---|---|
| *False* | ***True*** |
| *True* | ***False*** |

So the value of the condition is swapped.

With these operators in mind we can create a wide range of compound conditions where we can combine Conditional and logical operators, as we have seen already:

```
if (DateOfBirth > 1967 and DateOfBirth < 2002):
```

With this type of combinations, we can program complex sets of conditions.

#PythonMonday © Damian Gordon

# Getting Input from the User

**The Input Command**

If we return to our `DateOfBirth` program again:

```
DateOfBirth = 1985

 if (DateOfBirth > 1967):
     print("Date of birth is after 1967")
 else:
     print("Date of birth is not after 1967")
```

We can see that one drawback with this program is that the programmer has to set the `DateOfBirth` variable at the start of the program, but it would be better if we could ask the user to input their Year of Birth and the program could calculate whether or not that year was after 1967 or not.

In Python to get input from the user we use the `input()` function. So that function reads in whatever the user types, but it doesn't know whether the input is a number or a word, so if we are reading in a number we call tell Python by saying `int(input())`, and this converts whatever is being typed into an integer.

So to see it in action, we could do something as follows:

```
print("What is your Year of Birth:")
DateOfBirth = int(input())
```

So the first line is simply a message to the user telling them what to do, and the second line says to take whatever input the user types in, convert it into an integer, and store the result in the variable `DateOfBirth`. It's important to name a variable to store the value that was input from the user, it has to go somewhere.

We can do those two commands in one line as follows:

```
DateOfBirth = int(input("What is your Year of Birth:\n"))
```

And that will work in the exact same way

So to revisit our `DateOfBirth` program, this time using the `input()` command:

```
DateOfBirth = int(input("What is your Year of Birth:\n"))

 if (DateOfBirth > 1967):
     print("Date of birth is after 1967")
 else:
     print("Date of birth is not after 1967")
```

As we see, the only change needed is the manner in which the variable is read in, otherwise the code stays exactly the same. The difference is that in the original program it was the programmer who set the year, whereas in the new version of the program the user has the power to choose the year to be input, thus empowering the user, and giving them control of the system.

#PythonMonday © Damian Gordon

# Is it Odd or Even?

**Odd and Even Numbers**

How can we tell if a number is even or odd? For a human being, this is easy, but for a computer it's a bit harder. We know what odd and even numbers look like:

- **Even Numbers:** 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 …
- **Odd Numbers:** 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 …

And the rule is that even numbers can be divided evenly by 2, and odd numbers can't be divided evenly; or to put it another way, if you divide an even number by 2, the reminder is 0, and if you divide an odd number by 2, the remainder is 1.

**Division Remainder**

Is there a way to write a computer program that focuses just on the remainder of a division, and not the division result itself? As it happens there is, in Python there is a special type of division, called Division Remainder (as we saw on Page 7), that uses the "%" symbol, and returns the remainder result of a division. So if we use this form of division, and divide a number by 2, if the number is even, there will be no remainder, but if the number is odd, there is a remainder of one, For example:

| | | | | | |
|---|---|---|---|---|---|
| `print(9 % 2)` | `1` | `print(8 % 2)` | `0` | `print(7 % 2)` | `1` |
| `print(6 % 2)` | `0` | `print(5 % 2)` | `1` | `print(4 % 2)` | `0` |
| `print(3 % 2)` | `1` | `print(2 % 2)` | `0` | `print(1 % 2)` | `1` |

So we can ask the user to input a number, store it in the variable `InputNumber`, and then check if `(InputNumber % 2)` gives a result of 1, then it's odd, and if the result is 0, it's even. So the condition to check if it's odd could either be:

```
if (InputNumber % 2) == 1:
```

Or we could state the same condition as:

```
if (InputNumber % 2) != 0:
```

And they both give the same result (since the condition "equal to 1", is the same as "not equal to 0", in this particular case).

**Comments**

In a program, if we put the hash symbol ("#") at the start of a line, we are adding in a comment, which the Python interpreter will ignore, but it helps make the code clear for someone who is reading it. So the full code is as follows:

```
# PROGRAM IsOddOrEven:
InputNumber = int(input("Please input the number\n"))
if (InputNumber % 2) == 1:
    print(InputNumber, "is odd")
else:
    print(InputNumber, "is even")
# EndIf;
# END.
```

The comments have a PROGRAM name; an END to the IF, and an END program.

#PythonMonday © Damian Gordon

# Find the Biggest Number

**Bigger of Two Numbers**

If we wanted to write a program to take in two numbers from a user, compare them, and print out which one is bigger than the other, we could do the following:

```
# PROGRAM BiggerOfTwo:
FirstNumber = int(input("Please input the first value\n"))
SecondNumber = int(input("Please second the second value\n"))
if (FirstNumber > SecondNumber):
    print(FirstNumber, "is bigger than", SecondNumber)
else:
    print(SecondNumber, "is bigger than", FirstNumber)
# EndIf;
# END.
```

So if we typed in 22 and 33, it will print out the phrase:

```
33 is bigger than 22
```

And if we typed in 55 and 44, it will print out the phrase:

```
55 is bigger than 44
```

However, if we typed in 66 and 66, it will print out the phrase:

```
66 is bigger than 66
```

So the program doesn't really cover all eventualities, we need to check if the two values are the same to avoid this type of error. So we can first check if the input values are the same, and if they are, print out the value, otherwise we can check which one of the values is bigger than the other, in the same way as before:

```
if (FirstNumber == SecondNumber):
    print("Both the same number:", FirstNumber)
else:
    <CHECK WHICH IS THE BIGGER OF TWO>
    <IN THE SAME WAY AS WE OUTLINED ABOVE>
# EndIf;
# END.
```

So we put the original checking program into the else section (it must be indented):

```
# PROGRAM BiggerOfTwo-OrEqual:
FirstNumber = int(input("Please input the first value\n"))
SecondNumber = int(input("Please second the second value\n"))

if (FirstNumber == SecondNumber):
    print("Both the same number:", FirstNumber)
else:
    if (FirstNumber > SecondNumber):
        print(FirstNumber, "is bigger than", SecondNumber)
    else:
        print(SecondNumber, "is bigger than", FirstNumber)
    # EndIf;
# EndIf;
# END.
```

Note how the comments (with the hash symbol "#") make it easier to see alignment.

#PythonMonday © Damian Gordon

# What is Debugging?

*Thomas Edison, Isaac Asimov, J. Robert Oppenheimer, Grace Hopper*

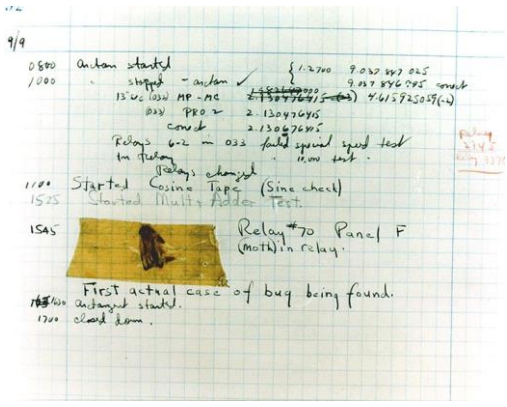
**Debugging is …**
Sometimes errors in programs are called “bugs”, so we have a special name for finding and fixing errors in computer programs, we call it “debugging” (in other words, taking the bugs out). These terms are not exclusive to computers, as far back as the 1870s, Thomas Edison uses the term in a letter, where he says: “*then that "Bugs"—as such little faults and difficulties are called—show themselves*”.

This term was used extensively in the 1930s and 1940s to describe *flaws* or *glitches* in mechanical devices. In 1944, the writer Isaac Asimov used the term in a fictional context describing potential errors in robots, in his short story "*Catch That Rabbit*", published in 1944: “*U. S. Robots had to get the bugs out of the multiple robots, and there were plenty of bugs, and there are always at least half a dozen bugs left for the field-testing*.”

That same year, on October 27th, in a letter from theoretical physicist, J. Robert Oppenheimer, when he was discussing the building of the first atomic bomb, he mentions, when discussing staff recruitment, that they are “*occupied in getting into operation and debugging*” the bomb.

On September 9th, 1947, computer developer Grace Hopper was tracing an error on the Harvard Mark II electromechanical computer.

One of the operators, William "Bill" Burke, found a moth trapped in a relay that was the cause of the error, so they taped the moth into the logbook, and recorded it as the first actual bug.




The notation reads:
"*First actual case of bug being found*."

# Programming is about Patience

**Fixing Programs Isn't Easy**

Now that you have been programming for a while you will have noticed that writing a program often isn't the hard bit, it's when you try to run it, and it gives you lots and lots of errors, and sometimes the error messages it gives you are too general to figure out exactly what the problem is. This is the hard part of programming, having the patience and persistence to review each line of your program to see if you can identify the problem. And often you have to stare at code for several minutes before you will, in a flash, figure out what's wrong. This is not easy, and it requires a lot of determination, because often when you have figured out (and fixed) one error, another one follows. So you have to type in your code very carefully, and review each line as you write it. But when you are finished writing a program, and go to run it, you need to accept that it will not run the first time, and when you fix the initial error, there may be another, and another, and another.

**Breathing**

Fixing programs can be stressful, and the longer you a working on a single program, the more frustrated you can get. This can lead to short-term thinking, where you move lines of code around at random in hopes that the program will fix itself. Try to avoid doing this, and try to avoid getting stressed by breathing. There are a variety of breathing techniques that can be used to calm down, including the following:

- ***Left Nostril Breathing***: As the name suggests, just close your right nostril off, and breath in and out through your left nostril slowly, with your eyes closed. This creates a calming effect in your nervous systems within minutes.
- ***7-2-11 Breathing****:* Breath in through your nose for 7 seconds, hold the breath for 2 seconds, and exhale through your mouth for 11 seconds. This takes a bit of practice, but after a few days of 4-8 sessions a day, you will master it.

**Take a Break**

One important trick to know is when to take a break; so if I am staring at an error and I can't figure it out, my rule of thumb is after 7 minutes I walk away from the computer and get a glass of water and stop thinking about it for 2-3 minutes. More often than not as soon as I return to the computer I know exactly the issue is, because I gave my unconscious mind time to work on it, and can fix it in no time.

**Cardboard Programmer**

Sometimes the easiest way to fix an error is to ask someone for help, but you will find that as soon as you say to someone "*Excuse me, can you help me with this problem…*" and before you have even outlined the problem, you know what the solution is, because you got a chance to think about it in a different way. In fact, you don't really need another person, just get a cardboard cutout and ask them for help.

#PythonMonday © Damian Gordon

# How to Find a Bug

**Debugging Approaches**

There are two parts to debugging, the locating and fixing of bugs. We'll look at locating the bugs first, so if the program runs but doesn't give us the results we are expecting, there is some error in the code that we need to find. The computer is only doing what it is told, so there must be a wrong instruction somewhere. There are a number of debugging approaches that can be taken to find that instruction:

- ***Brute Force Approach***: The is probably the most common approach to debugging, and it typically involves adding a number of `PRINT` statements throughout the program to determine the values of variables in different parts of the program, to see if it possible to locate the cause of the error. There are also tools that can be used in this approach, these include both tracing tools and debugging tools.
- ***Backtracking Approach***: The backtracking approach is exactly what it sounds like, you start at the end of the program where the results are being printed out from, and go backwards, manually reviewing each important line to see if it is correctly written, until the wrong instruction is found.
- ***Cause Elimination Approach***: This approach involves creating a list of possible causes (or hypothesis) for the error, and initial tests are carried out to eliminate each hypothesis. Of the ones that cannot be eliminated in the initial testing, further tests are carried out to eliminate more and more hypotheses, until there is only one cause left. The error is then located.

**One More Thing….**

This may be just me, but when I'm trying to debug a program, and I don't feel like I'm making progress; sometimes if I recite a verse of poetry, or part of a song, and I do that a couple of times, and it gives me the fortitude to continue and succeed. The two verses below are the ones I most commonly use, if it's any help.

*There's nothing you can do that can't be done*
*Nothing you can sing that can't be sung*
*Nothing you can say, but you can learn how to play the game*
*It's easy*

From the song "All You Need Is Love" by John Lennon and Paul McCartney (1967)

*Great bugs have little bugs upon their backs to bite 'em,*
*And little bugs have lesser bugs, and so ad infinitum.*
*And the great bugs themselves, in turn, have greater bugs to go on;*
*While these again have greater still, and greater still, and so on.*

"Siphonaptera" from Augustus De Morgan's *A Budget of Paradoxes* (1872)

#PythonMonday © Damian Gordon

## How to Fix a Bug

**Read the Error Message**
If the bug is producing an error message, read it carefully. Sometimes they can be unhelpful, but generally they can give you a clue as to what the issue might be, or at least what line number to start looking from. It can also be really useful to put the error message into Google and you will often find that someone else has experienced this error and has found a solution.

**Beware of Side-Effects**
Fixing a bug should be done very carefully, if the fix is done poorly it can introduce other errors into the program, and do more harm than good, so it's important to be careful when fixing bugs. American software engineer Tom Van Vleck outlined three simple questions that we should ask ourselves before making a fix:

1. *Is this bug (or a similar bug) likely to appear in another part of the code?*
   In many cases a programmer will use the same type of logic throughout the code, so if an error is found in one part of the program, it may be worth reflecting on whether or not there are other parts of the code that has similar functionality.

2. *What new error might be introduced into the program when fixing the error?*
   Before the error is fixed, it is a good idea to explore the design of the program, to check if the location where the bug was found is dependent on other parts of the program, in terms of sharing data structures or program logic. If there is a dependency (a coupling), then it is important to carefully check what the consequences could be of any changes made.

3. *What can be done to prevent this same bug from happening again?*
   This is the first step in creating a good Software Quality Assurance Process going forward. If there was a problem in the development process that caused the error to occur, fixing that issue with the process will prevent similar errors from occurring in the future.

**Other Terms for Bugs**
Depending on who is discussing bugs, they may use different terms to describe them, so for example, someone in IT Sales might call them *features*, whereas a tester might call them *issues*. Here are some other terms for bugs:

| | | | |
|---|---|---|---|
| ***Defects*** | ***Faults*** | ***Problems*** | ***Incidents*** |
| ***Anomalies*** | ***Inconsistencies*** | ***Variances*** | ***Failures*** |
| ***Mistakes*** | ***Exceptions*** | ***Errors*** | ***Side Effects*** |

This is a small sampling of the range of terms used for bugs.

#PythonMonday © Damian Gordon

# Common Issues with Input/Output

**Quotation Marks**

Quotation marks (“ ”) that are produced in a computer program editor (like IDLE) are slightly different from the ones you find in a Word document or in a PowerPoint presentation, so if you are copying code from either of these programs, check that the quotation marks are the right type:

| | |
|---|---|
| Here’s what they look like if they are written in Word or PowerPoint: | “ ” |
| Here’s what they look like if they are written in a program editor: | " " |

The computer won’t recognise the first set of quotation marks, so you need to delete them and retype them in doing [Shift] and [2].

**The Print Statement**

Two common errors that occur when people start to use the `print` statement in Python for the first time is that they either forget to include the quotation marks or the forget to include the brackets, as shown below:

| WRONG CODE | REASON |
|---|---|
| `print(Hello World)` | Left out quotation marks |
| `print “Hello World”` | Left out brackets |

So, in practice, the `print` statement should look as follows:

```
print(“Hello World”)
```

**The Input Statement**

Two common errors that occur when people start to use the `input` statement in Python for the first time is that they either include quotation marks around the input command (which they shouldn’t) or the forget to close the final brackets of the statement, as shown below:

| WRONG CODE | REASON |
|---|---|
| `InputVal = int(“input()”)` | Adding in quotation marks |
| `InputVal = int(input()` | Missing one of the brackets |

So, in practice, the `input` statement should look as follows:

```
InputVal = int(input())
```

**Indentation**

One of the most common issues with Python programs is the indentation of the statements, this helps the interpreter identify common blocks of code, so if you get an error, always check your indentation first.

#PythonMonday © Damian Gordon

# Common Issues using IF

**The Condition**

Two common errors that occur when people start to use the `if` statement in Python for the first time is that they either forget to add the colon (:) at the end of the statement or they capitalise the "i" in "if".

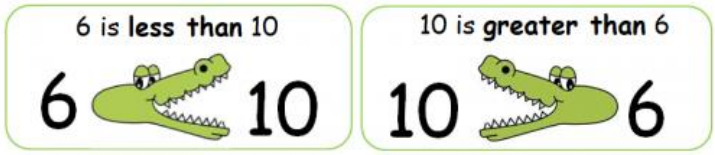
| WRONG CODE | REASON |
|---|---|
| `if (x > y)` | Missing the colon (:) at the end |
| `If (x > y):` | Capitalised the "i" in "if" |

So, in practice, the `if` statement should look as follows:

```
if (x > y):
```

**Greater Than or Less Than**

Sometimes people mix up "less than" and "greater than"; if that happens, this might help "*the crocodile always eats the bigger number*":




**Other Issues**

If we declare a variable as lowercase ("x") initially, then the computer won't recognise it if you change it to uppercase ("X"). Also, another issue is around indentation, make sure the "`else:`" statement is aligned with the "`if`".

| WRONG CODE | REASON |
|---|---|
| `x = int(input())`<br>`y = int(input())`<br>`if (X > y):` | "x" needs to be the same case:<br>`x = int(input())`<br>`y = int(input())`<br>`if (x > y):` |
| `if (x > y):`<br>`  print("X is bigger")`<br>`  else:`<br>`  print("Y is bigger")`<br>`# EndIf;` | No need to indent the "else":<br>`if (x > y):`<br>`  print("X is bigger")`<br>`else:`<br>`  print("Y is bigger")`<br>`# EndIf;` |

**Reflections**

It takes a bit of practice to get used of the precision required for programming, so don't get discouraged by making small errors at the start of your journey; you are learning a key skill, and it's worth taking the time, and being patient along the way.

#PythonMonday © Damian Gordon

# Types of Errors

**Syntax Errors**

"Syntax" is a term used to describe the rules of a language, and a syntax error is where we don't follow the rules of the language, so for example, in English (this is a speaking language or a "natural language") the phrase "*the cat sat on the mat*" follows the rules of the language, because in English we can have a noun ("cat") followed by a verb ("sat") followed by a preposition ("on"), followed by a noun ("mat"). So the chain *noun-verb-preposition-noun* follows the rules. However the phrase "*sat the cat the mat on*" is not legal because you the chain *verb-noun-noun-preposition* does not follow the rules of English grammar. In a programming language it's the same thing, the phrase "`X = 5`" is legal, but "**`X 5 =`**" isn't legal, and the computer won't understand what it means. Syntax errors are easy to locate because the computer will give an error when you run the program.

**Semantics Errors**

A semantic error is one where the syntax is correct, but it breaks some other rule in the programming language, and will not compile, so, for example, a type error:

```
x = "Hello, World!"
print(x+1)
```

Or using a variable before declaring it:

```
print(x)
x = "Hello, World!"
```

There are a number of typical semantic errors, and they usually revolve around a program that uses some resource before telling the program we want to use it.

**Logical Errors**

Logical errors don't give us any errors when we compile our programs, but they do give us the wrong answer, so if we wrote a program to calculate the area of a circle:

```
AreaOfACircle = 3.1416 * Radius
```

This is wrong, because the area of a circle is Pi * $R^2$, which should be written as:

```
AreaOfACircle = 3.1416 * Radius * Radius
```

So the first program will give us the wrong answer, but won't produce a compilation error. A common logical error on conditions of IF statements, where we mix up the "less than" ("<") and "greater than" (">"), so for example:

```
x = int(input("Input a value: "))
if (x > 5):
    print("X is less than 5")
# EndIf;
```

This program won't give a compilation error, but it will give the wrong answer.

#PythonMonday © Damian Gordon

# Using the WHILE Statement

**Repeating Commands**

Almost every program we write will require it to repeat certain sections of the code, so for example, if the program needs to print out the first one hundred numbers, i.e. 1, 2, 3, 4, 5, … 100, we could do it with one hundred print statements, but that would be a very time consuming process. Another simple example is a program that opens a text file and searches that file one line at a time for a particular phrase. In Python, one way to give the programmers the ability to repeat commands is to use the WHILE statement (or WHILE loop), so, for example if we were searching for someone's name in a phonebook:

```
while (the current name isn't the one we are looking for):
        check the next name
```

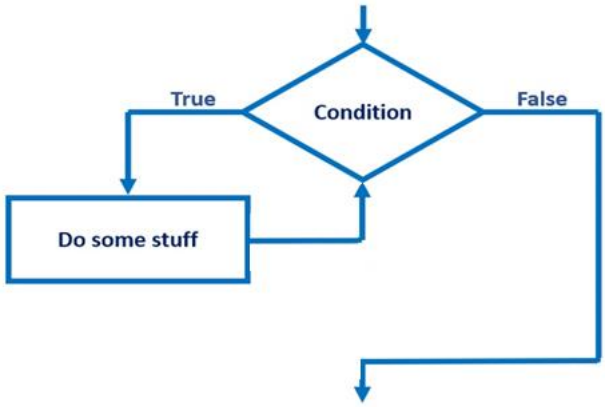
So, as we can see the WHILE statement has a *condition* like the IF statement, except that the instruction(s) inside the WHILE statement will keep being repeated ("keep looping") until the condition is found to be false.

***The WHILE Statement***

```
while (condition):
        do some stuff
```

Or as a picture, we can show the WHILE Statement, or WHILE loop, like the diagram below, where the statement starts with a "Condition" and if the condition is true it goes to "Do some stuff", and after it is finished, it returns to the condition and keeps checking that until the condition is found to be false, and then it skips onto the next statement:

***The WHILE Statement as a Flow Chart***




The diagram above is called a *Flow Chart*.

#PythonMonday © Damian Gordon

# Our First WHILE Statement

**The WHILE Statement**

***The WHILE Statement***

```
while (condition):
        do some stuff
```

The "condition" in a WHILE statement is exactly the same as the one in an IF statement, it has to be something that is either "True" or "False" (a Boolean), so for example, is today Monday will either be true or false. If the condition evaluates to "True" then the instructions in the WHILE block are run, if it evaluates to "False" then the loop exits, and the commands following the WHILE block are executed. Statements under the WHILE that are indented are part of the loop, and will be executed as long as the condition evaluates to True, but as soon as the condition is False, the next instruction that isn't indented will be run.

**Our First WHILE Program**

Below is our first program with a WHILE Statement. It works as follows, we create a variable called X, and set it to be the number 1, then we check if X is less than 6, and as long as it is, we will keep executing the loop.

***Sample WHILE Statement***

```
# PROGRAM Print1To5
X = 1
while (X < 6):
        print(X)
        X = X + 1
# EndWhile;
# END.
```

So in more detail, X is set to the number 1, then we check if X is less than 6, and it is, so we print out X (1), and then we add one onto X (making it 2). Then we return to the condition, check if it is still true (2 < 6), and it is, so we print out X (2), and then we add one onto X (making it 3). Then we return to the condition, check if it is still true (3 < 6), and it is, so we print out X (3), and then we add one onto X (making it 4). Then we return to the condition, check if it is still true (4 < 6), and it is, so we print out X (4), and then we add one onto X (making it 5). Then we return to the condition, check if it is still true (5 < 6), and it is, so we print out X (5), and then we add one onto X (making it 6). Then we return to the condition, check if it is still true (6 < 6), and it is not, so we stop the loop.

```
1
2
3
4
5
```

The condition is "less than 6", so when it gets to 6 the loop exits. Try this code, and then change the number to different values, try 11, 101, and 1001 to check this all makes sense.

#PythonMonday © Damian Gordon

# More on the WHILE Statement

**Summing Numbers**

If we wanted to add the numbers from 1 to 5, and store that value in a variable, we know how to count the numbers 1 to 5 already, we do it using a variable X that starts as one (1) and gets incremented each time the program executes that loop:

***Sample WHILE Statement***

```
# PROGRAM Print1To5
X = 1
while (X < 6):
        print(X)
        X = X + 1
# EndWhile;
# END.
```

To add the numbers together, we need a new variable, let's call it `SumTotal`, and we will set its starting value (also called the "*initial value*") to zero (0) and let's add the variable `X` onto the variable `SumTotal` during each execution of the loop. So the first time in the loop `X` is 1 and `SumTotal` is 1 (1 + 0 = 1), the next time around the loop `X` is 2 and `SumTotal` is 3 (2 + 1 = 3), the next time around the loop `X` is 3 and `SumTotal` is 6 (3 + 3 = 6), the next time around the loop `X` is 4 and `SumTotal` is 10 (4 + 6 = 10), the next time around the loop `X` is 5 and `SumTotal` is 15 (5 + 10 = 15), the loop then stops executing.

| X | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| SumTotal | 1 | 3 | 6 | 10 | 15 |

So the code below shows how we could write this program, there are just three extra lines: the first new line sets the initial value to zero (0), the second new line is inside the loop and it adds the current value of `X` to the current value of `SumTotal`, and the final new line prints out the value of `SumTotal`.

***Summing Numbers Using the WHILE Statement***

```
# PROGRAM Sum1To5:
X = 1
SumTotal = 0
while (X < 6):
        SumTotal = SumTotal + X
        X = X + 1
# EndWhile;
print(SumTotal)
# END.
```

So the output we will get from this program is:

```
15
```

And this is just counting to five (5), but we can use this to do any sum.

#PythonMonday © Damian Gordon

# Calculating Factorial

**Factorial (Getting the Product of Numbers)**
In mathematics the factorial of a number is that number multiplied by every number smaller than itself down to the number one, so, for example:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Or, in general terms, for any number N:

$$N! = N \times (N-1) \times (N-2) \times \ldots \times 2 \times 1$$

To write a Python program to calculate the factorial of a number, it is very similar to the program to sum all of the numbers up to a certain values, but you are multiplying instead of adding. So the first thing we need is a new variable, let's call it `ProductTotal`, and we will set its starting value (also called the "*initial value*") to one (1). We'll also have the counting variable `X`, that starts at one (1) and continues counting up to the desired value. So let's multiple the variable `X` by the variable `ProductTotal` during each execution of the loop. So the first time in the loop `X` is 1 and `ProductTotal` is 1 (1 * 1 = 1), the next time around the loop `X` is 2 and `ProductTotal` is 2 (2 * 1 = 2), the next time around the loop `X` is 3 and `ProductTotal` is 6 (3 * 2 = 6), the next time around the loop `X` is 4 and `ProductTotal` is 24 (4 * 6 = 10), the next time around the loop `X` is 5 and `ProductTotal` is 120 (5 * 24 = 120), the loop then stops executing.

| X | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| ProductTotal | 1 | 2 | 6 | 24 | 120 |

So the code below shows how we could write this program, it is very similar to the Adding program, but we are using multiplication instead of addition.

***Summing Numbers Using the WHILE Statement***

```
# PROGRAM Product1To5:
X = 1
ProductTotal = 1
while (X < 6):
        ProductTotal = ProductTotal * X
        X = X + 1
# EndWhile;
print(ProductTotal)
# END.
```

So the output we will get from this program is:

```
120
```

This is just calculating to five (5), but we can use this to do any product to get that factorial.

# Is it a Prime Number?

**What is a Prime Number?**
In mathematics a Prime Number is a number that is only evenly divisible by itself and the number one (1) with no remainder. So for example the number 7 is prime because it only divides evenly by the numbers [7, 1], and if you divide it by all of the numbers in between [6, 5, 4, 3, 2], it gives a remainder. To give an alternative example the number 9 is not prime because it divides evenly by the numbers [9, 1], but if you divide it by all of the numbers in between [8, 7, 6, 5, 4, 3, 2], you find that there is one number (three) that divides evenly into 9, and therefore 9 is not a prime number. So a general statement of how to check if a number is prime or not is as follows:

```
For any number N, it is a prime number if we divide it
by all the numbers less than it but greater than one
[N-1, N-2, … 3, 2], and they all give some remainder.
```

So the code below shows how we could write this program:

- we start by getting the number to be tested, and call it `CheckNum`. Then we create a variable for the divider (the *denominator*), and we call that `Countdown`, which starts at `CheckNum-1` and we keep taking one (1) off it each time the program loops until we get to two (2).
- To check if a division gives a remainder, we use the Remainder Division (%) operator, which will be zero (0) if it gives no remainder, and not zero if there is a remainder, so our IF statement is: `if (CheckNum % Countdown == 0):`
- Lastly, we have a Boolean variable called `IsPrime` that we set to `True` at the start of the program, which assumes the number is going to be prime, and then only can we set it to `False` if we are inside the WHILE loop and if the division returns a zero (0) remainder, which would mean that the number input is not prime.

***Summing Numbers Using the WHILE Statement***

```
# PROGRAM CheckPrime:
CheckNum = int(input("Please input value:"))
Countdown = CheckNum - 1
IsPrime = True
while (Countdown > 1):
    if (CheckNum % Countdown == 0):
        IsPrime = False
    # EndIf;
    Countdown = Countdown - 1
# EndWhile;
print(IsPrime)
# END.
```

So, the output we will get from this program is “True” if the number is prime, and “False” if the number isn’t prime.

#PythonMonday © Damian Gordon

# Calculating Fibonacci Numbers

**What are Fibonacci Numbers?**
Fibonacci (Leonardo Bonacci) was an Italian mathematician who published a book in 1202 called "Liber Abaci". In the book he discussed the growth of (idealised) rabbit populations and he proposed a sequence to model those populations as follows:

***1, 1, 2, 3, 5, 8, 13, 21, 34, …***

Where each number in the sequence is the sum of the two previous numbers, so to put it in more formal terms , for any number N: `Fib(N) = Fib(N - 1) + Fib(N - 2)`
In other words, any Fibonacci number can be calculated as the sum of the two previous numbers, so Fib(6) = Fib(5) + Fib(4), which is 8 = 5 + 3.

To figure out the number that the user wants to count to; we'll ask the user and save that in a variable called `Position` and we'll take one away from `Position` each time we are in the loop until we reach 1. Our program will start by setting two variables `Fib1` and `Fib2` to 1 and 0, and each time around the loop we calculate the Fibonacci number, `FibNumber`, by adding the two variables together. To calculate the next number in the sequence, we put the value of `Fib1` into `Fib2`, and the value of `FibNumber` into `Fib1`. The next time in the loop when we add `Fib1` and `Fib2` we will get the next element in the sequence:

| Position | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|
| Fib2 | 0 | 1 | 1 | 2 | 3 |
| Fib1 | 1 | 1 | 2 | 3 | 5 |
| FibNumber | 1 | 2 | 3 | 5 | End Loop |

So the code is as follows:

***Calculating Fibonacci Numbers***

```
# PROGRAM FibonacciNumbers:
Position = int(input("Please input value:"))
Fib1 = 1
Fib2 = 0
FibNumber = 1

while (Position >= 1):
    FibNumber = Fib2 + Fib1
    Fib1 = Fib2
    Fib2 = FibNumber
    Position = Position - 1
# EndWhile;
print(FibNumber)

#END.
```

So, the output we will get is the Fibonacci number that is in the position input.

# Common Issues using WHILE

**The Condition**

Two common errors that occur when people start to use the `while` statement in Python for the first time is that they either forget to add the colon (:) at the end of the statement or they capitalise the "w" in "while".

| WRONG CODE | REASON |
|---|---|
| `while (x > y)` | Missing the colon (:) at the end |
| `While (x > y):` | Capitalised the "W" in "while" |

So, in practice, the `while` statement should look as follows:

```
while (x > y):
```

**Loop Count**

Sometimes people are trying to count from 1 to 5 in the loop and have the condition on the loop that causes it to loop either one too few times or one too many times. It's a good idea to take the loop out of a bigger program and into a simple loop like the one on Page 37. So assuming that we have a loop counter X that starts at 1, and is incremented in each iteration of the loop, either of the following statements work:

```
while (X < 6):
while (X <= 5):
```

**Other Issues**

If we declare a variable as lowercase ("x") initially, then the computer won't recognise it if you change it to uppercase ("X"). Also, another issue is around indentation, make sure the body of the loop is indented from the "`while`".

| WRONG CODE | REASON |
|---|---|
| `x = int(input())`<br>`y = int(input())`<br>`while (X > y):` | "x" needs to be the same case:<br>`x = int(input())`<br>`y = int(input())`<br>`while (x > y):` |
| `while (x > y):`<br>`print("X is", x)`<br>`x = x + 1`<br>`# EndWhile;` | Need to indent the body:<br>`while (x > y):`<br>`  print("X is", x)`<br>`  x = x + 1`<br>`# EndWhile;` |

**Reflections**

It takes a bit of practice to get used of the precision required for programming, so don't get discouraged by making small errors at the start of your journey; you are learning a key skill, and it's worth taking the time, and being patient along the way.

#PythonMonday © Damian Gordon

# Origins of Open-Source Software

**The Origins**

Open-Source software is software that is usually developed in a collaborative public manner and is made available usually for free to anyone who wishes to use it, to change it, and even to incorporate it into new software products.

In the early days of software development (in the 1950s) this is the way programs were largely being developed, where programmers left copies of their code in public spaces, in the form of tapes or punch cards, for others to use. This may be because a lot of early software development was done in academic institutions where there is less focus on commercial considerations.

**The Hacker Ethic**

In the 1950s–1960s at Massachusetts Institute of Technology, college students who staged pranks were called "Hackers", and the term eventually became used more generally to describe people who got involved in constructive projects that were undertaken for the pleasure of being involved in them, including computer programming projects. These computer hackers developed an approach to life, a philosophy, an ethos, that they called the "Hacker Ethic". According to author Steven Levy in his 1984 book "Hackers: Heroes of the Computer Revolution", the six key principles of the Hacker Ethic are:

1. Access to computers—and anything which might teach you something about the way the world works—should be unlimited and total
2. All information should be free
3. Mistrust authority—promote decentralization
4. Hackers should be judged by their hacking, not bogus criteria such as degrees, age, race, sex, or position
5. You can create art and beauty on a computer
6. Computers can change your life for the better

**Open Source Software**

As more software corporations began to emerge, in the 1970s and 1980s, two distinct points of views emerged, on the one hand open source developers believe that sharing code means the new programmers can learn by reading lots and lots of existing code, and because anyone can look at open source programs, the majority of the flaws in those programs (the "bugs") will be discovered and corrected, producing highly reliable and coherent software. On the other hand, closed source software developers, or proprietary software developers, believe that commercial organisations that pay staff to develop software should be entitled to sell their programs for a fee. In fact, Bill Gates, one of the founders of Microsoft wrote an open letter to open source developers (sometimes called "hobbyists") where he told them that he thought they "*must be aware, most of you steal your software*".

#PythonMonday © Damian Gordon

# The Bill Gates Letter

February 3, 1976

An Open Letter to Hobbyists

To me, the most critical thing in the hobby market right now is the lack of good software courses, books and software itself. Without good software and an owner who understands programming, a hobby computer is wasted. Will quality software be written for the hobby market?

Almost a year ago, Paul Allen and myself, expecting the hobby market to expand, hired Monte Davidoff and developed Altair BASIC. Though the initial work took only two months, the three of us have spent most of the last year documenting, improving and adding features to BASIC. Now we have 4K, 8K, EXTENDED, ROM and DISK BASIC. The value of the computer time we have used exceeds $40,000.

The feedback we have gotten from the hundreds of people who say they are using BASIC has all been positive. Two surprising things are apparent, however, 1) Most of these "users" never bought BASIC (less than 10% of all Altair owners have bought BASIC), and 2) The amount of royalties we have received from sales to hobbyists makes the time spent on Altair BASIC worth less than $2 an hour.

Why is this? As the majority of hobbyists must be aware, most of you steal your software. Hardware must be paid for, but software is something to share. Who cares if the people who worked on it get paid?

Is this fair? One thing you don't do by stealing software is get back at MITS for some problem you may have had. MITS doesn't make money selling software. The royalty paid to us, the manual, the tape and the overhead make it a break-even operation. One thing you do do is prevent good software from being written. Who can afford to do professional work for nothing? What hobbyist can put 3-man years into programming, finding all bugs, documenting his product and distribute for free? The fact is, no one besides us has invested a lot of money in hobby software. We have written 6800 BASIC, and are writing 8080 APL and 6800 APL, but there is very little incentive to make this software available to hobbyists. Most directly, the thing you do is theft.

What about the guys who re-sell Altair BASIC, aren't they making money on hobby software? Yes, but those who have been reported to us may lose in the end. They are the ones who give hobbyists a bad name, and should be kicked out of any club meeting they show up at.

I would appreciate letters from any one who wants to pay up, or has a suggestion or comment. Just write me at 1180 Alvarado SE, #114, Albuquerque, New Mexico, 87108. Nothing would please me more than being able to hire ten programmers and deluge the hobby market with good software.

Bill Gates
General Partner, Micro-Soft

#PythonMonday © Damian Gordon

## The Cathedral and the Bazaar

**Cathedral**: A large church, with a highly structured architecture.

**Bazaar**: A marketplace, with different stalls selling different goods.

In 1997 software developer Eric S. Raymond presented a paper called “The Cathedral and the Bazaar” looking at two different philosophies of writing and releasing open source computer programs:

- The “Cathedral” approach is a highly structured one, where software (and source code) is made available one version at a time, as “releases”, and in between releases it is worked on by an exclusive group of programmers.
- The “Bazaar” approach is a far more open one, where the source code (and all changes made to it) is publicly available online, and any programmer can register and contribute to the code between releases.

Raymond released a book in 1999 also titled “The Cathedral and the Bazaar” which included his original paper, and further essays on software development. The central idea of the paper and book is that the more programmers who can see code in development, the more likely that the majority of the errors (“bugs”) in it will be discovered. He formulates this as "*given enough eyeballs, all bugs are shallow*”, and refers to it as “Linus's law”, named in honour of Linus Torvalds, the initial creator of the Linux operating system, who used a “Bazaar” approach in the development of that system. Although there are some disagreements in the software development community as to whether or not this is a universal law, most of the empirical studies seem to provide support for Linus's law.

He also identified some key lessons that can help ensure the success of an open source software development process. Programmers should be interested in the project, they should base their first draft on some existing code, but should not be afraid to completely redraft their code (and redraft it again). They should release their code early and often, and should treat their users and testers as co-developers. When bugs are found in software, if there is a large group of programmers working on it, at least one of them will know how to fix that bug in a simple way. If the overall project can be divided into distinct and independent sub-projects, all of the developers don’t need to communicate with each other, so there’s less confusion.

#PythonMonday © Damian Gordon

## Open-Source Projects

Because Python follows the open-source model of developing programs, there are a large number of on-going projects to develop tools in Python. You can participate in this process, all of the code for these projects are available to look at on a public repository (called *GitHub*), and you can look at the software that has been developed so far to see what large amounts of code looks like, and as time goes on you should consider contributing some code to one of these projects.

**OpenCV**
https://opencv.org/
Computer Vision is the study of developing software to help computers identify objects in images and videos, for example, inspecting bottles in a manufacturing production line, or facial recognition software. OpenCV was originally written in a combination of two programming languages (C and C++), but now provides interfaces to many other programming languages, including Python, so it's possible to develop features in Python and integrate them into OpenCV.

**Keras**
https://keras.io/
Artificial Intelligence is the study of developing software to help computers behave in a manner that appears intelligent, for example, medical diagnosis or automated characters in computer games. Keras was developed in Python in 2015 and provides a wide range of features that allows programmers to create complex and easily extendable artificially intelligent systems.

**Scikit-learn**
https://scikit-learn.org/stable/
Scikit-learn is also a suite of software for developing artificially intelligent systems, implemented mainly in Python. It works well with a lot of other Python libraries, and features various kinds of algorithms, including classification, regression and clustering.

**Django**
https://www.djangoproject.com/
Django is a framework to help develop database-driven websites. It is developed in such a way that it allows other people's code to be plugged into your code in a really simple way. There are thousands of packages available to extend the framework's original behaviour, that provide additional tools.

For more project ideas, check out this link:
https://www.upgrad.com/blog/python-open-source-project-ideas-topics/

## Copyleft and Free Software


**Copyright**
This is a type of intellectual property licencing that gives its owner the exclusive right to make copies of a creative work, usually for a limited time. Copyright is intended to protect the original expression of an idea in the form of a creative work.

**Copyleft**
This is a type of intellectual property licencing that gives people the right to freely distribute and modify content with the requirement that the same rights be preserved in derivative works created from that property.

**Creative Commons**
This is a collection of intellectual property licences (some of which are similar to copyleft licences) that enable the freer distribution of an intellectual property. This is used when an author wants to give other people the right to share, use, and build upon a work that has been created.


**Free Software Licenses**
Free-software licenses gives users the right to take a piece of software and modify and redistribute it. These licenses are granted by the rights-holder of the software and remove typical copyright restrictions by accompanying the software with a software license which grants rights.

**Free Software Foundation (FSF)**
The Free Software Foundation was founded by Richard Stallman on October 4, 1985, to support the free software movement, which promotes the universal freedom to study, distribute, create, and modify computer software. It supports several free software licenses, meaning it publishes them and has the ability to make revisions as needed.

#PythonMonday © Damian Gordon

## Notable Legal Copyright Cases

Some important cases that have had a real impact on the notion of whether software can be free or not, and whether software can be created to freely share other forms of intellectual property include the following:

**Apple Computer, Inc. v. Franklin Computer Corp.**
The Franklin Computer Corporation created a computer known as the *Franklin Ace 1000*, which at first appeared to be a clone of Apple Computer's *Apple II*. A clone is a hardware or software system that is designed to function in exactly the same way as another system. However, Apple figured out that significant portions of the Franklin system was copied directly from the Apple II, so they filed a lawsuit in 1983. Franklin admitted that it had copied Apple's software but argued that because Apple's software existed only in binary form, and not in printed form, it could be freely copied, and some of the software didn't have copyright notices on it. The court found in favour of Franklin, but on appeal the court found for Apple, and found that operating systems were also copyrightable. The parties settled.

**A&M Records, Inc. v. Napster, Inc.**
The Recording Industry Association of America (RIAA) took a lawsuit against Napster, Inc. in 2001 for copyright infringement. Napster was an online service that allowed people to share files with each other (called a peer-to-peer (P2P) file sharing service) focusing on sharing digital audio files. Napster also provided a central server that indexed connected users and files available on their machines, creating a searchable list of music available across Napster's network. Napster claimed that people were sharing files to create backups of music they had already purchased, or sampling music before they were going to purchase it. The Court disagreed, and an injunction was issued ordering Napster to prevent the trading of copyrighted music on its network.

**Authors Guild, Inc. v. Google, Inc.**
Book authors and publishers from the Authors Guild and the Association of American Publishers took copyright cases against Google between 2005 and 2015. The cases centred on the legality of the Google Book Search tool that transforms printed copyrighted books into an online searchable database through scanning and digitization. However, many authors and publishers had expressed concern that Google had not sought their permission to make scans of their books still under copyright. Google argued that they were scanning these books under fair use, and they worked with the litigants in both suits to develop a settlement agreement which was rejected by 2011. In late 2013 judgement was given in favour of Google, dismissing the lawsuit and affirming the Google Books project met all legal requirements for fair use. This judgement was upheld by the Appeals Court in 2015.

#PythonMonday © Damian Gordon

# Contributing to Python

**How to Become a Contributor**

You shouldn't feel you need to jump straight into writing program code to add new features to the Python language, there are a few things you can do to help develop the language without writing any code, for example:

- The most important thing you can do in use the language, just spend time writing computer programs and noting any issues with language.
- Send those issues on to the Python community.

**How to Contribute to Programming of the Python Language**

To contribute to the Python programming language, the first thing you need to do is to create a login account, by going here. Once you have completed the login process you can get involved in the community in a range of ways that are listed here.

**The Python Issue Tracker**

An easy way to start contributing is to look at the *Python Issue Tracker*. If you think you have found an issue (or bug) with Python, you can go to the Tracker here, and report that bug, and if you have found an issue specifically with the documentation, there is a sub-list that can be found here.

**Helping with the Documentation**

Python provides clear and detailed documentation, which is created and updated by community members. The documentation covers things like the features of the current version of Python, as well as how to set up Python on your computer, and what features are offered by the Python Libraries, it is available here. The Python Issue Tracker sub-list on documentation (mentioned above) that can be found here has a range of issues that can vary from things like typos, to there being unclear documentation, to items that are completely lacking documentation. Other useful links include the following:

- Helping with document quality
- Documentation Style Guide
- Translating the documentation into other languages

**Helping with the Programming**

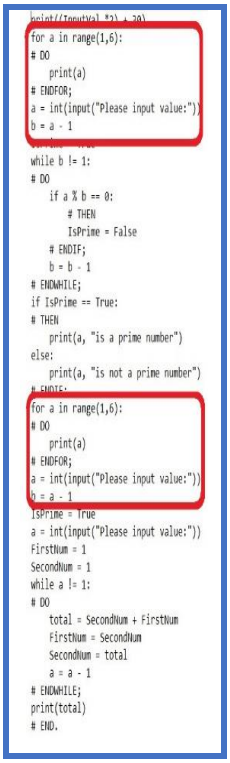
If you are interested in helping with the programming of the Python language, it is first better to start reading other people's code to see what they have written and how they write it. Following that the python site has a great page on Getting Started which explains how to install all the computer programs you need to help with the programming of the Python language. The next step is to figure out Where to Get Help, including mailing lists and IRC. Other useful links include the following:

- Running and Writing Tests
- Following Python's Development
- Git Bootcamp and Cheat Sheet

#PythonMonday © Damian Gordon

## What are Functions and Methods?

### Functions and Methods




Imagine if we had a large Python program that has several sections of the code repeated throughout the program. It would be good if there was some way we could wrap up the frequently used commands into a single package, and instead of having to rewrite the same code over and over again, we could just call the package name instead. In Python we usually call these packages *functions*.

Functions are normally designed to accomplish a single, specific task, for example, check if a number is odd or even. In this case the function would be called with its package name and the number to be checked, and the function would return back to the main program whether the number is odd or even (as a Boolean).

Functions are sometimes called *Methods*, but only if they are defined as part of a larger structure called a *Class*. We won't be learning about classes in detail for the moment, but it is worth mentioning that classes are part of an approach to programming called *Object-Oriented design*, which focuses on defining the main functions of a program as general structures called *objects*.

The first thing we need to do is decide on a name for the function, use the **`def`** command with that name. Following the name are a pair of parenthesis (which may or may not include the names of values (parameters) that need to be included in the function). Next the full set of commands that are part of the function are included (and they are indented), followed by an optional **`return`** command, which lets the function pass a value back to the main program. We normally finish the function with a commented **`END`** command.

***FUNCTION Format***

**`def`** *FunctionName*(*InputParamters*):
*do some stuff*;
**`return`** *ReturnValue*
**`# END`** *FunctionName*.


Functions are great because they allow us to reduce the amount of code in a program, and also allow us to focus on *what* the overall program is doing instead of concentrating on *how* the overall program is doing it (this is called *abstraction*).

## Our First Function

**Sample Function**

As we've seen, a function can do the following:

1. Take in some values from the main program inside brackets,
2. Do some activity using the usual programming commands,
3. Return a result back to the main program.

A simple example might be to determine whether or not a number is odd or even. To make things easy for ourselves, we should choose whether we are testing if the number is even or testing if the number is odd. We can pick either, but in the example below we'll check if the number is even, and we'll call it `IsEven`. If we find the number is even, we will return `True`, and if the number is odd, we will return `False`. So the number that goes into the function is going to be called `InputNumber`, and we check if it divides evenly into the number 2, if it does we set a variable to `True`, and if not, we set it to `False` (the variable is called `ReturnValue` here). Finally, we return that result back to the main program:

***IsEven FUNCTION***

```
def IsEven(InputNumber):

    if (InputNumber % 2) == 0:
        ReturnValue = True  #it's even
    else:
        ReturnValue = False #it's odd
    # EndIf;

    return ReturnValue
# END IsEven.
```

**The Main Program**

The main program that calls this function should be contained in the same Python file as the function (after the function), and if we put the following in:

```
print(IsEven(4))
```

Python will tell the `IsEven` function to put the number "4" into the `InputNumber` variable, and we will get the following output:

```
True
```

And if put the following in:

```
print(IsEven(3))
```

Python will map the number "3" onto `InputNumber` variable, and we will get:

```
False
```

So Python looks at the value in the brackets when the function `IsEven` is called and will map that onto the `InputNumber` variable.

#PythonMonday © Damian Gordon

## Calling the Function

### The Role of a Function

If we look at the `IsEven` function again:

*IsEven FUNCTION*

```
def IsEven(InputNumber):

    if (InputNumber % 2) == 0:
        ReturnValue = True  #it's even
    else:
        ReturnValue = False #it's odd
    # EndIf;

    return ReturnValue
# END IsEven.
```

We note that the `IsEven` function only does one thing, it takes in a value and checks if it's even or not, it doesn't deal with asking the user for a number or printing a message out to the user. This is the key philosophy behind the design of functions, *they should do one thing, and do it well*. So a function shouldn't try to do two or three things, or even half a job, it should always strive to do just one thing, as well as possible.

### Calling the Function

When the main program uses the name of the function, we say "*the program is calling the function*", and we saw an example of that, where we say the following:

```
print(IsEven(4))
```

We can also call the function in a way that includes getting input from the user, and prints an answer back to them. We do this using the `Input` function to get a value from the user, then we call the `IsEven` function as part of the condition of an IF statement, and if the `IsEven` function returns `True` we print out the number with the message that the number is even, otherwise we print out the number is odd:

```
GetNumber = int(input("Input number:\n"))

 if (IsEven(GetNumber) == True):
    print(GetNumber, "is an even number")
 else:
    print(GetNumber, "is an odd number")
 # Endif;
```

Note that the name of the variable being passed into the function from the main program is called `GetNumber`, but when the value for that variable is received by the function, it is stored in a new variable with a different name `InputNumber`. This is normal programming practice, and it means the programmer who writes the main program doesn't need to know anything about how the function is written.

#PythonMonday © Damian Gordon

## Divisible By Function

**Divisible by 3**

We could write a program to check if a number is evenly divisible by the number 3 by taking the `IsEven` function and instead of checking if there is a remainder when dividing by 2, we check if there is a remainder by dividing by 3 instead:

***IsDivisibleBy3 FUNCTION***

```
def IsDivisibleBy3(InputNumber):
    if (InputNumber % 3) == 0:
        ReturnValue = True  # Divisible by 3
    else:
        ReturnValue = False # Not divisible by 3
    # EndIf;

    return ReturnValue
# END IsDivisibleBy3.
```

And the main part of the program could say something like:

```
print(IsDivisibleBy3(15))
```

And we would get the following output:

```
True
```

**Divisible by N**

If we wanted to make the program more general, we could use it to check if a number is evenly divisible by any other number, all we need to do is pass a second value into the function, in this case `N`, and doing a division of the `InputNumber` by `N` (we call in input values "parameters", and in this case, there are two parameters):

***IsDivisibleByN FUNCTION***

```
def IsDivisibleByN(InputNumber, N):
    if (InputNumber % N) == 0:
        ReturnValue = True  # Divisible by N
    else:
        ReturnValue = False # Not divisible by N
    # EndIf;

    return ReturnValue
# END IsDivisibleByN.
```

And the main part of the program would have to take in two values, for example:

```
print(IsDivisibleByN(15, 2))
```

We will get the following output:

```
False
```

And if we did `print(IsDivisibleByN(15, 3))` we would get back `True`.

#PythonMonday © Damian Gordon

## Prime Number Function

**Prime Number Function**

We've already seen how to check if a number is prime or not, we just divide it by all the numbers less than it and greater than one, and if any of them divide evenly into the number (i.e. gives no remainder) then we know that the number isn't prime. To take the program we have and make it into a function, the first thing we do it to get the number under investigation, passed into the function as a parameter, in this case called `InputNumber`. We do the computation exactly the same way as before, and then we return an answer back to the main program, in this case a Boolean value, where `True` means the value is prime, and `False` means that the value is not prime:

***IsPrime FUNCTION***

```
def IsPrime(InputNumber):
    Countdown = InputNumber - 1
    ReturnValue = True
    while (Countdown > 1):
        if (InputNumber % Countdown == 0):
            ReturnValue = False
        # EndIf;
        Countdown = Countdown - 1
    # EndWhile;
    return ReturnValue

# END IsPrime.
```

The code in the main part of the program is almost exactly the same as previous examples, except that this time it calls the function `IsPrime`. So the main program deals with the Input and Output to the users (sometimes called "I/O"), and the calculations and computations are done by the function:

```
GetNumber = int(input("Input number:\n"))

if (IsPrime(GetNumber) == True):
   print("It's a prime number")
else:
   print("It's not a prime number")

# Endif;
```

And if the number inputted is 11, then we will get the following output:

```
It's a prime number
```

We note again that it is alright if the name of the variable that passes the value into the function is different from the variable that receives that value in the module.

#PythonMonday © Damian Gordon

## Fibonacci Function

**Fibonacci Function**

For the Fibonacci function we can take the Fibonacci code that we've seen previously, and convert it into a function by adding a function name (with a `def` statement) and a return statement. We also remove all `print` statements from the function, as we prefer the main program deals with all of the user (I/O) Input/Output (where possible). Unlike the previous functions, the Fibonacci program won't return a Boolean, instead it returns an integer, and that integer is the Fibonacci number that is at the position indicated by the input parameter `InputNumber`:

***Fibonacci FUNCTION***

```
def CalcFib(InputNumber):
    Fib1 = 1
    Fib2 = 0
    FibNumber = 1

    while (InputNumber >= 1):
        FibNumber = Fib2 + Fib1
        Fib1 = Fib2
        Fib2 = FibNumber
        InputNumber = InputNumber - 1
    # EndWhile;
    return FibNumber

# END CalcFib.
```

As before, the main part of the program, which is code that can follow the function in the same file deals with the Input and Output (I/O) to the users:

```
GetValue = int(input("Please input value: "))
print("Fibonacci Number is", CalcFib(GetValue))
```

And if the number inputted is 10, then we will get the following output:

```
Fibonacci Number is 55
```

If we wanted to print out the first 20 Fibonacci number we can do it by changing the main program from a simple I/O request into a print statement in a loop, as follows:

```
FibCount = 1
while (FibCount < 21):
    print(FibCount, "Fib is", CalcFib(FibCount))
    FibCount = FibCount + 1
# EndWhile;
```

This calls the function 20 times with the numbers 1 to 20 in the variable `FibCount`.

#PythonMonday © Damian Gordon

## Common Issues with Functions

**Function Name**

When we are calling a function, it's really important to get the name of the function right, this might seem obvious, but it's amazing how often people get the name wrong, and it's a big problem because the computer won't know which function we are talking about unless we get the name right (it can't guess). This is particularly a problem if the name of the function is made up of several words, so, for example, we remember the function `IsDivisibleBy3`, it could go wrong like this:

| WRONG CODE | REASON |
|---|---|
| `print(DivisibleBy3(15))` | Needs the correct name: `print(IsDivisibleBy3(15))` |

**Input Parameters**

When we are calling a function it's really important to know how many parameters (input values) we need to pass into a function, because if we pass in too many parameters, or too few parameters, it will give us an error. So, for example, we remember the function `IsDivisibleByN` takes in two parameters that are numbers, so `print(IsDivisibleByN(15, 2))` returns `False`, because 2 doesn't divide evenly into 15, but `print(IsDivisibleByN(15, 3))` returns `True`. The function takes in two parameters, and it could go wrong like this:

| WRONG CODE | REASON |
|---|---|
| `IsDivisibleByN(15)` | Too few parameters |
| `IsDivisibleByN(15, 2, 4)` | Too many parameters |

Another common issue is when we pass in the wrong type of parameters, so for example, the `IsDivisibleByN` takes in two parameters that are numbers, and it could go wrong if something other than numbers are input as parameters:

| WRONG CODE | REASON |
|---|---|
| `IsDivisibleByN(@, &)` | The parameters are characters |
| `IsDivisibleByN(False, True)` | The parameters are Boolean |

**The Return Value**

The two functions `IsDivisibleBy3` and `IsDivisibleByN` both return a Boolean value (either `True` or `False`), so when we call those functions we need to make sure that we are checking for the right return type:

| WRONG CODE | REASON |
|---|---|
| `if (IsDivisibleBy3(15) > 7):` | The return value is a Boolean so you can't compare it to a number (7). |

So it's important to understand what values are going in and out of a function.

#PythonMonday © Damian Gordon

# Introduction to Testing

**Testing is …**

When we write computer programs, it's really important that we check to make sure the programs are working correctly. We've already talked about "debugging", that is where we find out that the program has an error in it when we try to run it, and then have to fix it, so "testing" is where we carefully examine the program to check if it is working correctly, and to check if it is doing what we want it to do.

**Testing Inputs**

If programs take in values from users, it's important to test those programs on a variety of different inputs to see how they react, and if they can detect the difference between valid and invalid inputs. So, I usually try:

| Expected Input | Test Values |
|---|---|
| **Number** | I usually start off with a few small numbers (1, 2, 3), then I try a few bigger numbers (99999, 999999), then I try zero (0) often programs don't consider a zero input, and then I'd try a few negative numbers (-1, -2, -3, -9999999). Next, I'd try a few characters (A, B, C, @), and finally I'll try the <Space> character. |
| **Characters** | I'd try some uppercase letters (A, B, C), then I'll try some lowercase letters (a, b, c), then I'll try some numeric characters (1, 2, 3), and a few other characters (@, #, %). Following these I'd try a few strings of characters (AAAA, BBBB, CCCC, @@@@), and finally I'll try the <Space> character. |
| **Date** | I try today's date, then twenty years ago, then I try a few simple valid dates (10/10/1010, 11/11/1111), and some invalid dates (22/22/2222, 00/00/0000) and then the <Space> character. |

**Testing Outputs**

We can also check if our programs give the right outputs, so, for example, if we write a program to double a number, but instead of multiplying the input number by 2, we accidently multiplied it by 3, our program wouldn't give an error, but it's still wrong:

```
# PROGRAM DoubleNumber:
OurValue = int(input("Please input value: "))
print("Double that number is", OurValue * 3)
# END.
```

So, if we typed in the number 4, we know we're expecting 8, but we'd get the wrong answer, 12. It's really important to check programs to see if they are consistently giving the correct answer by having a set of known inputs and expected outputs.

**Reflections**

These are two simple examples of testing (checking inputs and output), but testing includes a wide range of approaches to check if a program is working correctly.

#PythonMonday © Damian Gordon

# A Simple Function to Test

**A Simple Function to Test**

Let's imagine we wrote a function that needs to ask the user to either agree or disagree to something, and all they have to do is to select "y" or "n", as follows:

***Yes or No FUNCTION – Version 1***

```
def YN_Question():
    answer = input("Do you wish to exit (y/n)? ")
    if (answer == "y"):
        print("You selected YES")
    else:
        print("You selected NO")
    #EndIf;

# END YN_Question.
```

And we would call the function as follows (by adding the following to the bottom of the function in the same file):

```
YN_Question()
```

**Testing the Function**

The problem with this function is that it works well if we type in "y" or "n" , but if we tested this function with something else, like "g" or "d", the function would say:

```
You selected NO
```

Because the check on the function is to see if they typed in "y", and the assumption is that if they didn't type in "y", they must have typed in "n", but it is possible that they typed in some other value, therefore, we should check for that, as follows:

***Yes or No FUNCTION – Version 2***

```
def YN_Question2():
    answer = input("Do you wish to exit (y/n)? ")
    if (answer == "y"):
        print("You selected YES")
    else:
        if (answer == "n"):
            print("You selected NO")
        else:
            print("INVALID INPUT:", answer)

# END YN_Question2.
```

Now this version of the function will check for "y" or "n", and if it's neither of those two it will print a message "INVALID INPUT:" followed by the value typed in.

#PythonMonday © Damian Gordon

# Adding More to the Tested Function

**Adding More to the Tested Function**

Having tested the function and found that it needs to check if the user types in something other than "y" or "n" instead of just informing them that they typed in an incorrect value, sometimes we might want to reject any incorrect input, and keep prompting them until they type in one of those two values. If we want to do that, we need to do the following: read in the input, and have a loop that will keep executing as long as the value that has been typed in is both not equal to "y" and not equal to "n". This loop will keep running until a "y" or "n" is typed in, and when it is, the function can check which of the two values were input, as follows:

***Yes or No FUNCTION – Version 3***

```
def YN_Question3():

    answer = input("Do you wish to exit (y/n)? ")

    while (answer != "y") and (answer != "n"):
        print("INVALID INPUT:", answer)
        answer = input("Please input y or n:")
    # EndWhile;

    if (answer == "y"):
        print("You selected YES")
    else:
        print("You selected NO")
    # EndIf;

# END YN_Question3.
```

And we would call the function as follows (by adding the following to the bottom of the function in the same file):

```
YN_Question3()
```

**Testing the Function**

When we run the function, we will first get:

```
Do you wish to exit (y/n)?
```

And if the value input was "p", the function would say:

```
INVALID INPUT: p
```

And if the value input was "x", the function would say:

```
INVALID INPUT: x
```

And finally if we input the value "y", the function would say:

```
You selected YES
```

So this function will keep printing out the message "INVALID INPUT:" followed by the value typed in, until the user types in a "y" or an "n".

#PythonMonday © Damian Gordon

## Some Principles of Testing

**Fundamental Principle of Testing**

- The computer scientist Edsger Dijkstra once famously said that "*Program testing can be used to show the presence of bugs, but never to show their absence!*" (EWD249), in other words checking that a program works with various inputs and under different conditions might uncover some errors (bugs) in a program, but because we can't check every possible input and circumstances, it's not possible to prove that a program will work in all scenarios using testing alone.

**Other Principles of Testing**

- *Exhaustive testing is not possible, but optimal testing is necessary*: As noted in the first principle it is not possible to test every possible input and circumstances, therefore we have to prioritise the parts of the program we think are most likely to cause errors and also prioritise the parts of the program that would be most serious (or risky) if they caused an error.
- *Bugs like to hang out with other bugs*: This is sometimes called "Defect Clustering" in other words if we find an error (bug) in part of a large program, we should have a look at the code in that region of the program, because there might be further errors near the original. This is like the Pareto Principle, 80% of the errors are found in 20% of the program.
- *We have to regularly change our approach to testing*: If we use the same tests on all of our programs, they will only locate certain kinds of errors. We need to change our approach for different programs even if they have a similar function, so we can add new tests onto our existing ones, revising existing tests, or just change the whole testing process.
- *Testing isn't just about checking if the code works*: The sooner testing begins, the sooner errors can be located, we should not wait for the programs to be written to start our testing process, we can test the design of our programs before the coding. We should test all of our assumptions, as well as the code.

**BIOGRAPHY: Edsger W. Dijkstra**

Dijkstra was born in Rotterdam on 11th May 1930 and died in Nuenen on 6th August 2002. He is one of the most influential and important people in the history of computer science. His contributions cover areas including compiler design, operating systems, distributed systems, program design, program verification, software engineering, graph algorithms, and the philosophical foundations of computer science. His testing paper, "*On the Reliability of Programs*", (EWD303) is considered a classic in the field of testing.

#PythonMonday © Damian Gordon

# Eras of Software Testing

**Eras of Software Testing**

Software testing has evolved and improved over the past 75 years, and has seen many changes as a result of changes in technologies, processes and perspectives on testing. Presented below is a table outlining some of the key eras of testing based on David Gelperin and Bill Hetzel's paper "The Growth of Software Testing" published in 1988. I've added in the last three eras myself, based on various textbooks.

| Era | Description |
|---|---|
| 1945-1956 ***Debugging-Oriented*** | This was at the start of the history of programming, and is sometimes called the "Code-and-Fix" era, where there was no testing process, programmers fixed code as they found errors. |
| 1957-1978 ***Demonstration-Oriented*** | This was the first time there was a clear distinction between testing and debugging; and the testing focussed on ensuring that the program was doing everything it was supposed to do. |
| 1979-1982 ***Destruction-Oriented*** | During this era, the goal was to see what inputs would cause the programs to fail, for example. if we put a text value in a numerical field, or we put in a date of birth after today's date. |
| 1983-1987 ***Evaluation-Oriented*** | This era focused on testing as part of a larger quality assurance process; where it was acknowledged that large software systems would inevitably have some bugs in them, but to minimise the number of bugs to a specified rate. |
| 1988-2000 ***Prevention-Oriented*** | In this era, testers were expected to have a very good understanding of the systems that they were testing, and to know which parts of the code would be more difficult to test. |
| 2001-2003 ***Methodology-Oriented*** | Testing gained a new prominence and importance in this era with the advent of software development methodologies that put testing at their core, including Test-Driven Development. |
| 2004-2013 ***Automation-Oriented*** | In this era, large software testing tools were developed to help the testers do their job by eliminating some of the repetitive tasks, as well as creating large sets of input data, and inputting that data, and checking that the outputs are as expected. |
| 2014-To Date ***Intelligence-Oriented*** | Finally, we are now in an era where the testing tools are augmented by artificial intelligence that can help the tester figure out what tests are best for each part of the software. |

Note: These dates are all approximate, and, in reality, these eras don't fit into tidy little boxes, so in practice the eras overlapped significantly, but for the sake of understanding the key evolutions in software testing, this is a really good model.

#PythonMonday © Damian Gordon

# Creating a Testing Function

**Creating a Testing Function**

Let's remember the program we saw before that is supposed to double a number, but that it doesn't do it correctly. If we were to present it as a function, we could have it taking in an input value, multiply that value by 2 (but we've accidently multiplied it by 3), and returning the answer back, as follows:

***Double Number FUNCTION***

```
def DoubleNumber(InputValue):

    TheResult = InputValue * 3    #This is wrong
    return TheResult

# END DoubleNumber.
```

And we could call the program as follows (by adding the following to the bottom of the program in the same file, or by typing it directly into the command prompt):

```
DoubleNumber(3)
```

And unfortunately, the answer we would get is:

```
9
```

To help automatically test if a function is working, sometimes it is easier to create a new function (usually with the same name as the original function, but proceeded with `test_`) to check if the function is working. In the simple case of doubling a number, it might not be necessary, but if the function had several inputs, or you wanted to run a lot of tests, it can be useful. Our example would look as follows:

***Test Double Number FUNCTION***

```
def test_DoubleNumber():

    OurCheck = DoubleNumber(3)
    if (OurCheck == 6):
        print("This test was passed.")
    else:
        print("We better check DoubleNumber!!!")
    #EndIf;

# END test_DoubleNumber.
```

And we could call the program as follows:

```
test_DoubleNumber()
```

Note that if the result we get is correct (so if `DoubleNumber(3)` did give 6) , we just say "This test is passed.", we don't say "The function is working" because we don't know, we'd have to do a lot more testing on the function to be sure. So, we'd have to test it on zero, negative numbers, really big numbers, decimals, blank inputs, characters, etc. before we could start to have confidence in the function.

#PythonMonday © Damian Gordon

# A Better Testing Function

**A Better Testing Function**

Our current testing function is very simple, it only tests if one value works:

***Test Double Number FUNCTION***

```
def test_DoubleNumber():

    OurCheck = DoubleNumber(3)
    if (OurCheck == 6):
        print("This test was passed.")
    else:
        print("We better check DoubleNumber!!!")
    #EndIf;

# END test_DoubleNumber.
```

We can make it a lot more flexible by changing the value being fed into the `DoubleNumber` function (3) and the expected output (6) into variables that are passed into the function as parameters, as follows:

***Test Double Number FUNCTION – Version 2***

```
def test_DoubleNumber2(TestValue, ExpectedOutput):

    OurCheck = DoubleNumber(TestValue)
    if (OurCheck == ExpectedOutput):
        print("This test was passed.")
    else:
        print("We better check DoubleNumber!!!")
    #EndIf;

# END test_DoubleNumber2.
```

And now we can easily test a wide range of values, as follows:

```
test_DoubleNumber2(3, 6)
test_DoubleNumber2(0, 0)
test_DoubleNumber2(-5, -10)
test_DoubleNumber2(0.1, 0.2)
```

And if we get an error in any or all of the tests, we can keep changing the `DoubleNumber` function until it is working properly, and get the following output:

```
This test was passed.
This test was passed.
This test was passed.
This test was passed.
```

So it's important to remember that we may have to run and rerun the same test many times. If we get an error, then we will have to fix the function being tested, and rerun the tests again to check if it is fixed, and keep rerunning for all changes.

#PythonMonday © Damian Gordon

# What is an Array?

**Making a List, and Checking it Twice**

Let's imagine we were working in a school, and we have to keep a record of the ages of all the children in each class. We could do this by creating a variable for each student, as follows:

```
StudentAge1 = 9
StudentAge2 = 10
StudentAge3 = 9
StudentAge4 = 8
StudentAge5 = 10
StudentAge6 = 10
StudentAge7 = 9
StudentAge8 = 11
```

This is a bit cumbersome, and we have a simpler way of creating and managing a collection of variables, called an ***array***. We can think of it as a list of values which has a single name. So, we can declare an array as follows:

```
StudentsAges = [9, 10, 9, 8, 10, 10, 9, 11]
```

Meaning that we've created an array with the name `StudentsAges`, and we've initialised the array with the values of the students' ages. We can give the array any name we want (excluding the Python keywords) and any of the usual types, including integers, real numbers, strings, characters, and Booleans. We can picture the array as follows:

| StudentsAges | 9 | 10 | 9 | 8 | 10 | 10 | 9 | 11 |
|---|---|---|---|---|---|---|---|---|

==========

**The Pill Box Analogy**

I like to think of an array this way, we can either have a collection of several individual pill boxes, where it's easy to misplace one of them or to overlook one of them; or we can have a single box with all the pills in it, and we can see what we have taken and when we have to take our pills more easily:

| *Individual pill boxes* | *A single box with all the pills* |
|---|---|
|  |   |

The key point being, let's collect all the related variables into a single container.

#PythonMonday © Damian Gordon

# Elements of an Array

**Elements of an Array**

If we declare an array with eight values (or "elements") as follows:

```
StudentsAges = [9, 10, 9, 8, 10, 10, 9, 11]
```

Then we have a collection of integer variables, and if we want to talk about a particular value in the list, we address them by number, starting at zero (0). So the first element of the array is: `StudentsAges[0]` and if we say the following:

```
print(StudentsAges[0])
```

We will see the following written on the screen:

```
9
```

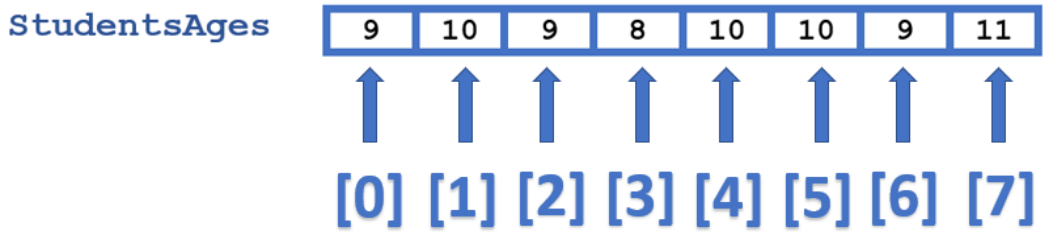
and if want to print out the second element in the array, we say the following:

```
print(StudentsAges[1])
```

We will see the following written on the screen:

```
10
```

And so on, if we want to picture this array with its address (or "index") values:

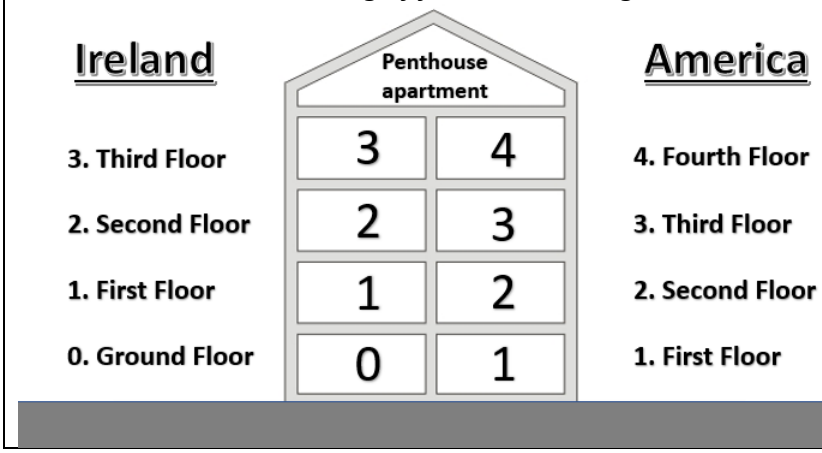



==========

**The Hotel Analogy**

I like to think of the elements of an array this way, in Ireland when you go to a hotel, the floor you first enter is the bottom floor, and it's called the "Ground Floor". The next floor up is called the "First Floor", and so on. If we think of the Ground Floor, as "Floor Zero", then the labelling of floors is the same as the indexing of an array:

*Labelling of floors in buildings*




The key point being, in Python we start an array at value zero (0).

#PythonMonday © Damian Gordon

## Changing Values in an Array

**Printing the Array**

If we have our array as follows:

```
StudentsAges = [9, 10, 9, 8, 10, 10, 9, 11]
```

If want to print out the full array, we say the following:

```
print(StudentsAges)
```

We will see the following written on the screen:

```
[9, 10, 9, 8, 10, 10, 9, 11]
```

**Updating the Array**

If we want to change the value of one of the elements of the array, we can say:

```
StudentsAges[2] = 111
```

And then when we print out the array again:

```
print(StudentsAges)
```

We will see the following written on the screen:

```
[9, 10, 111, 8, 10, 10, 9, 11]
```

If we want to add one to the first element of the array, we could say:

```
StudentsAges[0] = StudentsAges[0] + 1
```

And then when we print out the array again:

```
print(StudentsAges)
```

We will see the following written on the screen:

```
[10, 10, 111, 8, 10, 10, 9, 11]
```

So, the first value was 9 and now has become 10.

If we want to subtract one from the last element of the array, we could say:

```
StudentsAges[7] = StudentsAges[7] - 1
```

And then when we print out the array again:

```
print(StudentsAges)
```

We will see the following written on the screen:

```
[10, 10, 111, 8, 10, 10, 9, 10]
```

So, the last value was 11 and now has become 10.

If we want to multiply the first element of the array by one hundred, we could say:

```
StudentsAges[0] = StudentsAges[0] * 100
```

And then when we print out the array again:

```
print(StudentsAges)
```

We will see the following written on the screen:

```
[1000, 10, 111, 8, 10, 10, 9, 10]
```

So, the first value was 10 and now has become 1000.

#PythonMonday © Damian Gordon

# Using a WHILE Statement with an Array

**Using the WHILE Statement**

If we want to print out all of the elements in an array, another way of doing it is to create a WHILE statement to count from zero (0) to seven (7), and print out that element of an array:

***Print Elements using a WHILE Statement***

```
# PROGRAM PrintArray
X = 0
while (X < 8):
    print(StudentsAges[X])
    X = X + 1
# EndWhile;
# END.
```

And we will get the following:

```
9
10
9
8
10
10
9
11
```

**Updating Elements in an Array**

If we wanted to add one (increment) to each element of any array, all we have to do is this:

***Incrementing Elements using a WHILE Statement***

```
# PROGRAM IncrementArray
X = 0
while (X < 8):
    StudentsAges[X] = StudentsAges[X] + 1
    X = X + 1
# EndWhile;
# END.
```

This will add one (1) onto each element of the array, and we will get the following:

```
[10, 11, 10, 9, 11, 11, 10, 12]
```

If we change the line after the WHILE statement to say the following:

```
StudentsAges[X] = 0
```

This will set all elements of the array to zero, and we will get the following:

```
[0, 0, 0, 0, 0, 0, 0, 0]
```

#PythonMonday © Damian Gordon

# Using the FOR Statement

**Using the FOR Statement**

Python has a special type of loop to allow us to visit elements in an array (or any other collection of variables), and it's called the FOR statement, so it works in a similar way to the WHILE statement, but is more compact. All we have to do is create a variable that will store each element in the collection one at a time, and that will run all the commands inside the loop for each value, until there are no more elements in the collection. So, the general form of the FOR statement is as follows:

***The FOR Statement***

```
for (variable) in (collection):
        do some stuff
```

So. if we have our array as follows:

```
StudentsAges = [9, 10, 9, 8, 10, 10, 9, 11]
```

We could print out all of the elements of array as follows:

```
for AnyNameAtAll in StudentsAges:
    print(AnyNameAtAll)
# EndFor;
```

And we will get the following:

```
9
10
9
8
10
10
9
11
```

If we wanted to search to check if a particular value is in an array, we could do the following:

```
for AnyNameAtAll in StudentsAges:
    if (AnyNameAtAll == 8):
        print("Number 8 has been found")
    # Endif;
# EndFor;
```

This will visit each element of the array, and for each occurrence of the number eight (8), the program will print out a message:

```
Number 8 has been found
```

So if our program needs to visit each value in an array, or some other collection, the FOR statement is a nice, clear way of doing it.

#PythonMonday © Damian Gordon

# The FOR Statement with Strings

**The FOR Statement with Strings**

A String is a collection of characters enclosed in double quotes (""), we have seen a lot of examples of Strings already, for example, every time we use the PRINT statement:

```
print("Hello, World!")
```

If we want to use the same message a few times, it might make more sense to store the String in a variable and then print out that variable:

```
Greeting = "Hello, World!"
print(Greeting)
```

Python treats a String as being the same as an array of characters, so we can access the characters the same way we access elements of an array:

```
print(Greeting[0])
```

And we will get the following:

```
H
```

And we could print out all of the letters of the String as follows:

```
for EachLetter in Greeting:
    print(EachLetter)
# EndFor;
```

And we will get the following:

```
H
e
l
l
o
,

W
o
r
l
d
!
```

If we wanted to search to check if a letter is in a string, we could do the following:

```
for EachLetter in Greeting:
    if (EachLetter == "l"):
        print("The letter l has been found")
    # Endif;
# EndFor;
```

This will visit each element of the String, and for each of the three occurrences of "l", the program will print out a message: "The letter l has been found".

#PythonMonday © Damian Gordon

# The FOR Statement with the RANGE Function

**The FOR Statement with the RANGE Function**

So far, we have seen the FOR statement being used to display the values in an array or string, but if we want to update those values we can use the RANGE function to do that very easily. The RANGE function generates a sequence of numbers as follows:

```
x = range(8)
```

will create an array with the numbers 0,1,2,3,4,5,6,7.

So we can print out X as follows:

```
for count in x:
    print(count)
# EndFor;
```

And we will get the following:

```
0
1
2
3
4
5
6
7
```

So, if we have our array as follows:

```
StudentsAges = [9, 10, 9, 8, 10, 10, 9, 11]
```

If we wanted to add one (1) to each element of the array, we could do the following:

```
for Count8 in range(8):
    StudentsAges[Count8] = StudentsAges[Count8] + 1
# EndFor;
print(StudentAges)
```

And we will get the following:

```
[10, 11, 10, 9, 11, 11, 10, 12]
```

By default, the range function starts at zero (0), but we can start at any number by doing the following:

```
x = range(4, 8)
for count in x:
    print(count)
# EndFor;
```

And we will get the following:

```
4
5
6
7
```

So the RANGE command can take two parameters: `range(start, stop)`

#PythonMonday © Damian Gordon