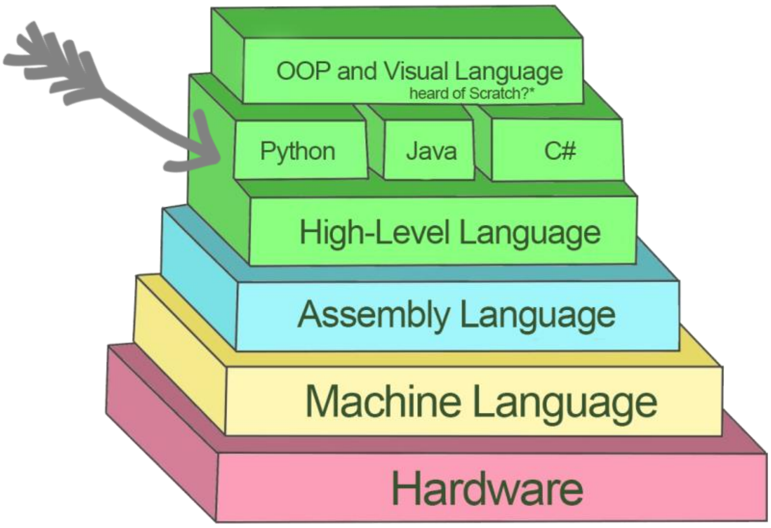
Python Basics Short Course

Hello, and thanks for starting on this accelerated Python basics course. I'm very happy to hear you're looking to launch into the world of Python and am pleased to be aiding your lift-off!

Before we start, what is Python? Well just the same as we have regular (natural) languages such as English, Japanese, and Arabic; computers too have their own set of complex languages. These languages are what we call **programming languages.**

But computers don't understand text and characters? True, they don't. Computers deal strictly with numbers (specifically binary[1]). That said, we have a hierarchy of language types designed to speak between one another. Like when we use an interpreter/friend to speak to someone else for us in another language.

Python is a high-level language. This simply means it is closer to natural language and it uses regular words and letters (not just numbers). Our code written in Python is compiled into an assembly language and then assembled for the machine to understand its meaning (this has all been worked on by many great minds over many years)[2].



Great, now what will I learn about Python? This course will provide you with a solid foundation and understanding of core programming ideas whilst providing useful resources and support to take your learning further in future.

Specifically, we will cover:

- **Installation** (**setting-up** on your computer)
- **Syntax** (the language's **writing structure**)
- **Variables**
- **Data Types**
- **Collections**
- **Conditionals**
- **Loops**
- **Functions (Bonus)**

Course length: 3-4 hours

The main aim is to build a strong foundation, in a short space of time. As with everything, the more effort you put in the faster you will improve and can move further into learning Python. As this is an introductory course, we will focus on procedural programming. This is a style similar to how we read text, in that our code is written from top to bottom and is read by the computer from top to bottom. Read the conclusion and bonus sections for advice on natural progression into other styles (formally called paradigms) of programming.

* For more on the Scratch visual language check out https://scratch.mit.edu/about

[1] For more on binary numbers check out https://arith-matic.com/notebook/binary-numbers

[2] For more on why we have all these languages check out https://www.scienceabc.com/innovation/why-are-there-so-many-programming-languages.html

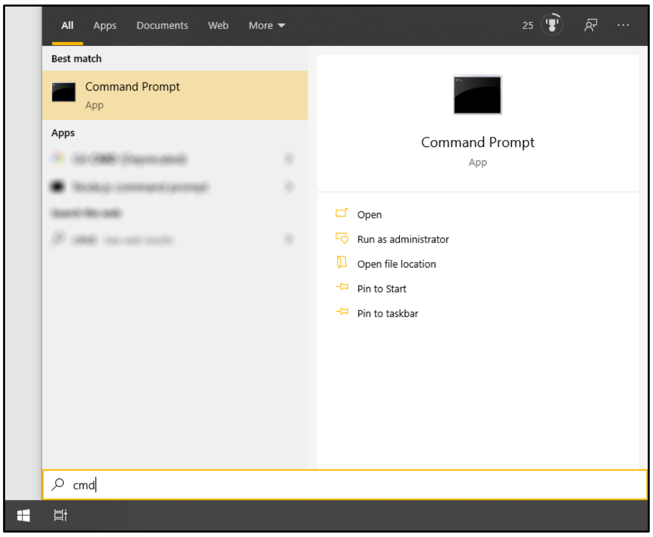
Installation and set-up:

- **Windows (*continue below*)**
- **macOS (*page 2*)**
- **Linux (*page 3*)**

If you do not want to download and run Python on your machine – you can use this online tool to execute Python code. **Skip to page 7 :)**

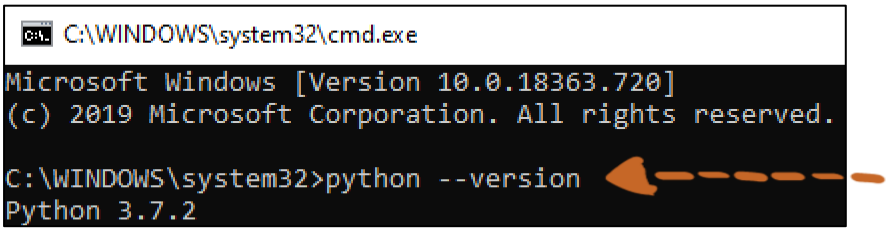
Windows installation process:

Sadly, Windows doesn't come with Python pre-installed. But to check if you may already have it, go to the Windows menu and Search **"cmd"** before opening (running) **Command Prompt.exe**



In versions of Windows earlier than Windows10 it will be called cmd

Now, type **`python --version`** and if you see a similar result to below, congratulations! You already have Python! However you will want to update the version you have to at least Python 3.0.[i]



```
Microsoft Windows [Version 10.0.18363.720]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>python --version
Python 3.7.2
```

Installing on Windows

Go to http://python.org/downloads/ and click any version of Python above 3.0 (so any number higher). Download the installer, and run it. **Make sure to check the "*Add Python to PATH"* box when you do this.**[ii]

After you are done installing, repeat the **`python --version`** command in a new cmd window (don't use the same window as earlier). If this time it works, you're finished!

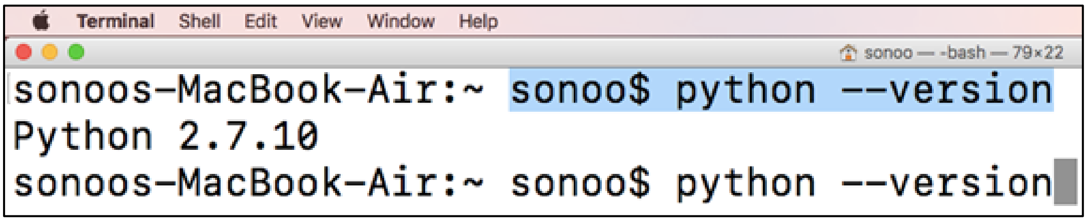
If it hasn't worked, it could mean it is not in your PATH. See the endnotes for how to do this manually.

macOS installation process:

Python comes pre-installed on most macOS systems, you will need a version 3.0 or higher to follow along the course (some things are different with older versions and 3 is the major version).

Firstly, press **Command (⌘) + Spacebar** to open Spotlight, then type **"Terminal"**. Now double click the result to open (run) it.

Now, type **`python --version`** and if you see a similar result to below, congratulations! You already have Python! However you will want to update the version you have to at least Python 3.0.



```
sonoos-MacBook-Air:~ sonoo$ python --version
Python 2.7.10
sonoos-MacBook-Air:~ sonoo$ python --version
```

Installing on macOS

The easiest way to install Python 3 on macOS is using a package called Homebrew. So we'll need to get that first. Homebrew depends on something called Xcode from Apple so we can just use the following command to install Homebrew:

```
$ xcode-select --install
```

Click through the confirmation dialogs that pop up. Next, install Homebrew with the following command:

```
$ ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Make sure you include a space between `curl -fsSL` and the URL. The `-e` in this command tells Ruby (the programming language Homebrew is written in) to execute the code that's downloaded here. You should only run commands like this from sources you trust.

To confirm that Homebrew was installed correctly, run the following command:

```
$ brew doctor
Your system is ready to brew.
```

This output means you're ready to install Python packages with Homebrew.

To install the latest version of Python 3, enter the following:

```
$ brew install python3
```

Confirm we are using python3, check as before except now use **`python3 --version.`**

If these steps do not provide some *Python 3.X.X* output, then use the reference in the endnotes for an alternative macOS Python installation tutorial.[iii]

Linux installation process:

Python comes pre-installed with almost all Linux distributions. However we'll need version 3.0 or higher for this course. To check the current version you open a **Terminal**. Now type **`python --version`**.

```
$ python --version
Python 2.7.6
```

Your system may already have Python 3.0 or higher *as well*, we can check using

`python3 --version`

```
$ python3 --version
Python 3.5.0
```

If you don't have Python 3 or need to update, you can use your particular package system - usually apt (Advance Package Tool) commands - to install. Below is a traditional Ubuntu example:

Remember you will need superuser privileges – so **`sudo`** *is needed.*

```
$ sudo apt install software-properties-common
```

This will allow us to add PPA (Personal Package Archive) repositories to our system.

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
```

Deadsnakes is a PPA with newer releases than the default Ubuntu repositories.

```
$ sudo apt-get update
```

Refresh the package lists again

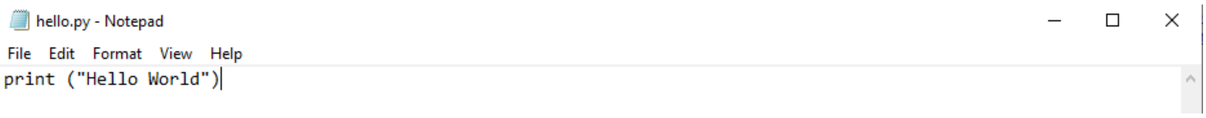
```
$ sudo apt-get install python3.8
```

Now we can install Python 3.8 (or 3.5 etc.)

Text editor installation:

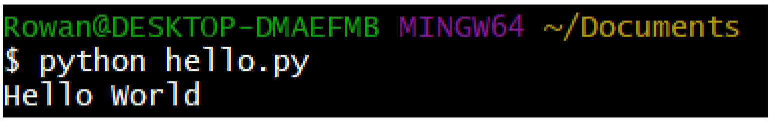
For any programming, you need somewhere to program right? Well yes, but its not as complex as you think! Generally we all have Notepad or some sort of place to write text – and you can code using that! But obviously we might make mistakes and it is very labourious to type every small thing out (think how things like Microsoft Word complete words for you and underline mistakes). All of the programs which we use for typing text into are known as **Text Editors**[iv]**.** On Windows and Mac a program called Python IDLE (Integrated Development and Learning Environment) comes with Python by default. We could use it write Python programs, however I think using something more versatile for your future programming makes more sense.

Just to quickly show the issue with using notepad, here's how we would write a program that says "Hello World" using Microsoft Notepad and a regular cmd terminal to run it:



```
print ("Hello World")
```

Be sure to save your file as name_of_file.py so your system knows it's a Python file.



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents
$ python hello.py
Hello World
```

We'd need to make a change, save, open a terminal, go to the folder it's in and then finally command it to be executed (run it).

Some editors are more capable and naturally offer more help than others, specifically a very advanced editor is **Visual Studio Code** (**VSCode** not to be confused with *Visual Studio*). It's an editor developed and supported by Microsoft and easily allows us to do all we need (it can work on any text the same as notepad might).

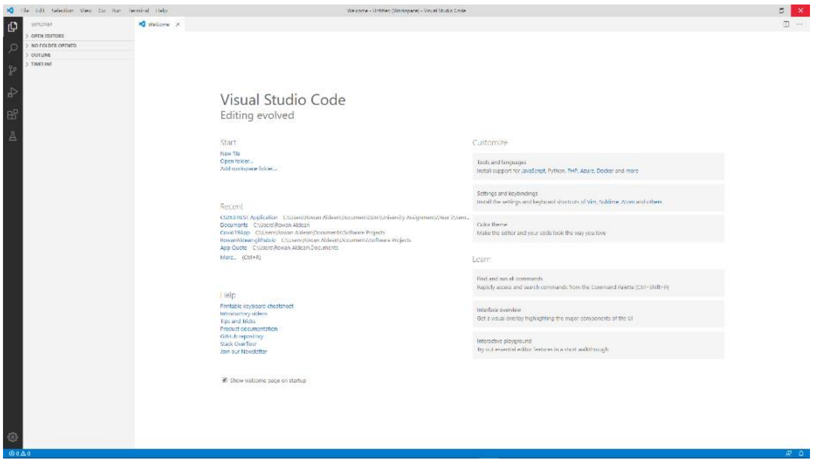
Use one of the installation guides below for your relevant operating system:

- **Windows (**https://code.visualstudio.com/docs/setup/windows**)**
- **macOS (**https://code.visualstudio.com/docs/setup/mac**)**
- **Linux (**https://code.visualstudio.com/docs/setup/linux**)**

To learn more about VSCode overall you can find more documentation at:

https://code.visualstudio.com/docs

Here is the layout and welcome page you should see upon starting VSCode:



Install support for JavaScript, Python, PHP, Azure, Docker and more

Install the settings and keyboard shortcuts of Vim, Sublime, Atom and others

Make the editor and your code look the way you love

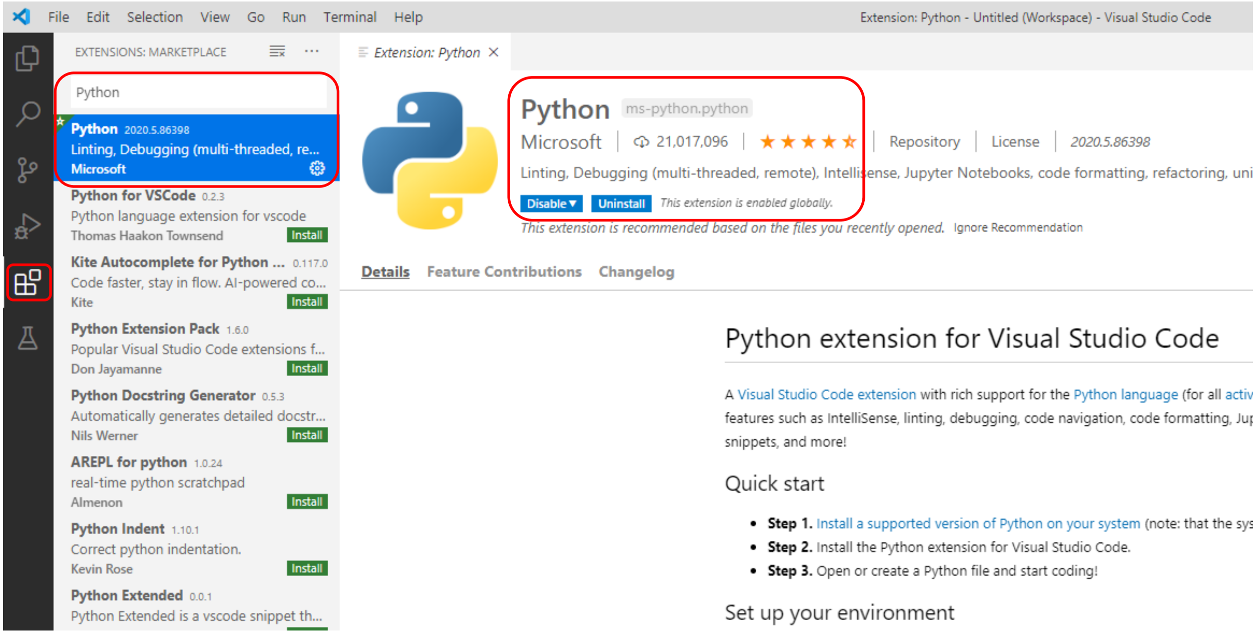
Rapidly access and search commands from the Command Palette (Ctrl+Shift+P)

Get a visual overlay highlighting the major components of the UI

Try out essential editor features in a short walkthrough

Configuring VSCode for Python:

Now we have VSCode we'll set things up to make our lives a little easier in future. First head over to the 4 squares icon (the "Extensions" tab) and search for **"Python".** Then click install on the extension by Microsoft!



Python

Python 2020.5.86398
Linting, Debugging (multi-threaded, re...
Microsoft

Python for VSCode 0.2.3
Python language extension for vscode
Thomas Haakon Townsend

Kite Autocomplete for Python ... 0.117.0
Code faster, stay in flow. AI-powered co...
Kite

Python Extension Pack 1.6.0
Popular Visual Studio Code extensions f...
Don Jayamanne

Python Docstring Generator 0.5.3
Automatically generates detailed docstr...
Nils Werner

AREPL for python 1.0.24
real-time python scratchpad
Almenon

Python Indent 1.10.1
Correct python indentation.
Kevin Rose

Python Extended 0.0.1
Python Extended is a vscode snippet th...

Python ms-python.python
Microsoft | 21,017,096 | ★★★★★ | Repository | License | 2020.5.86398
Linting, Debugging (multi-threaded, remote), Intellisense, Jupyter Notebooks, code formatting, refactoring, uni
This extension is enabled globally.
This extension is recommended based on the files you recently opened.

Python extension for Visual Studio Code

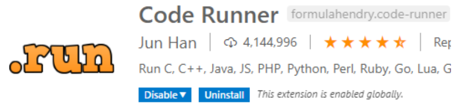
A Visual Studio Code extension with rich support for the Python language (for all activ
features such as IntelliSense, linting, debugging, code navigation, code formatting, Jup
snippets, and more!

Quick start

- **Step 1.** Install a supported version of Python on your system (note: that the sys
- **Step 2.** Install the Python extension for Visual Studio Code.
- **Step 3.** Open or create a Python file and start coding!

Set up your environment

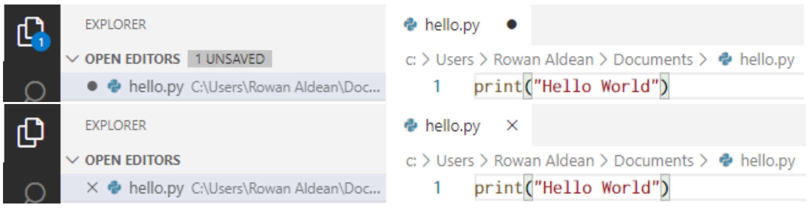
Optional: install the Code Runner extension by Jun Han to more easily run our programs.



Code Runner formulahendry.code-runner
Jun Han | 4,144,996 | ★★★★★ | Rep
Run C, C++, Java, JS, PHP, Python, Perl, Ruby, Go, Lua, G
This extension is enabled globally.

Now head over to the pages icon (the "Explorer" tab) and select **File -> New File.** Inside this new file we can write any text. But we wish to write Python so to allow the editor to understand we'll select **File -> Save As** and save it somewhere as **"hello.py".**

We can now do as we did in Notepad and write **`print("Hello World")`**. To quickly save our files we can use **Ctrl + S (Mac: ⌘ + S).** An unsaved file will have a black dot appear next to its name in the editor. When saved this should disappear.


EXPLORER
OPEN EDITORS
1 UNSAVED
● hello.py C:\Users\Rowan Aldean\Doc...
hello.py ●
c: > Users > Rowan Aldean > Documents > hello.py
EXPLORER
OPEN EDITORS
× hello.py C:\Users\Rowan Aldean\Doc...
hello.py ×
c: > Users > Rowan Aldean > Documents > hello.py

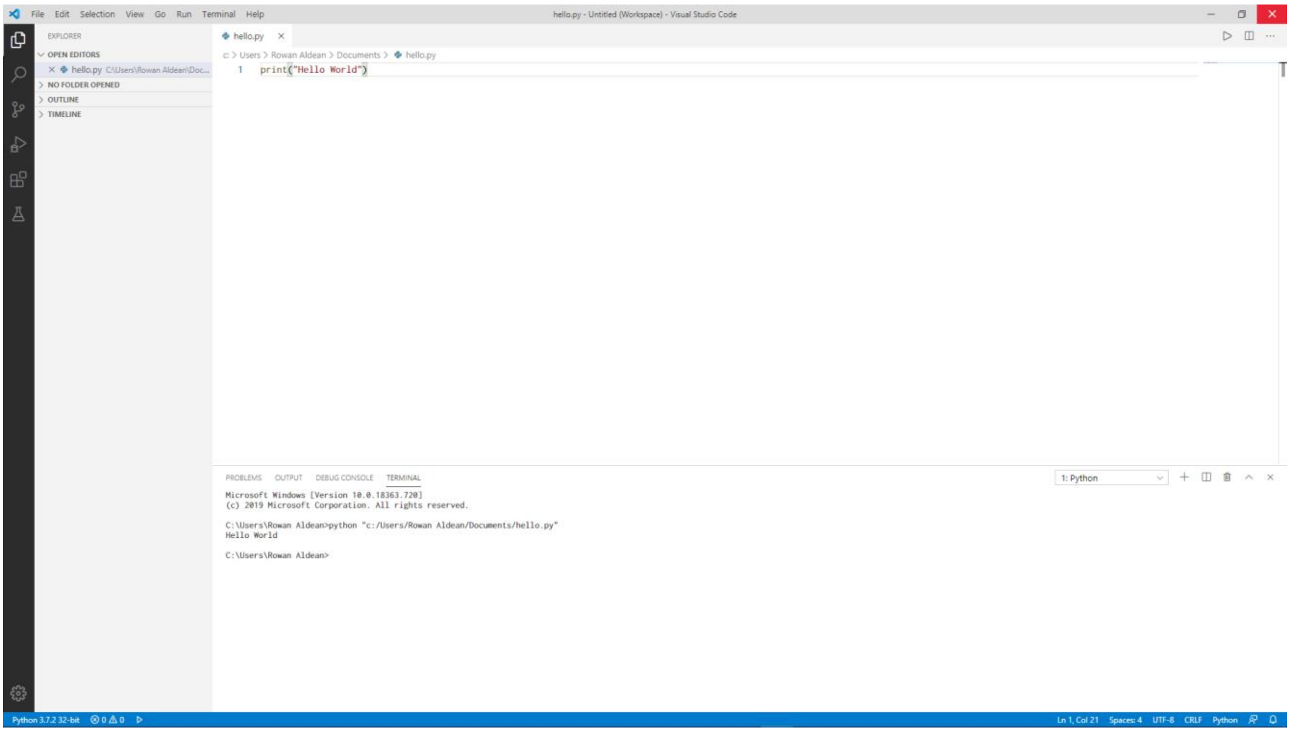
```
print("Hello World")
```

```
print("Hello World")
```

The beauty of editors like VSCode also means we can quickly "run" our programs inside the same window. The terminal is integrated – so we can choose to select **Run -> Run Without Debugging** or with the help of our Python extension we can easily **Right Click -> Run Python File in Terminal** or even quicker the Code Runner extension lets us **Ctrl + Alt + N (Mac: ^ + ⌥ + N).** To run the current file.

You can bind these actions to different keyboard shortcuts.[v] I strongly suggest you do ;-)

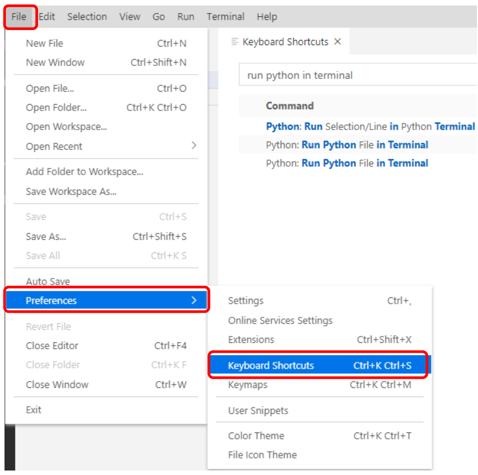
Below is the result of the prior steps – notice how everything is easy and in one place, unlike if we had used Notepad or some other very basic editor. VSCode will use the default terminal for your system; in this case that is the Windows cmd (Command) Prompt.



```
print("Hello World")
```

```
Microsoft Windows [Version 10.0.18363.720]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Rowan Aldean>python "c:/Users/Rowan Aldean/Documents/hello.py"
Hello World

C:\Users\Rowan Aldean>
```

Keyboard Shortcuts (Quick Tutorial)

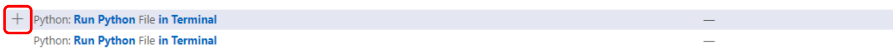


run python in terminal

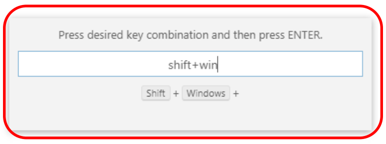
Command

Python: Run Selection/Line in Python Terminal
Python: Run Python File in Terminal
Python: Run Python File in Terminal

Head to **File (Mac: Code) -> Preferences -> Keyboard Shortcuts** and search for the **"run python in terminal"** action. You can now press the **"+"** symbol to set a keyboard shortcut for this action.



Python: Run Python File in Terminal —
Python: Run Python File in Terminal —



Press desired key combination and then press ENTER.

shift+win

Contents

Python Syntax

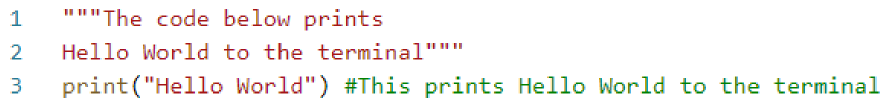
Syntax formally means “the arrangement of words and phrases to create well-formed sentences in a language”. As you’d imagine, every language (programming languages too) has their own syntax.

Comments

We’ve seen how we can write some words to the computer using the **`print`** keyword, but now we want to explain what this is doing to someone else who might read our code (or ourselves looking back on it in future) we’ll need to add some comments.

We use the **#** symbol to comment on 1 single line. Comments are not read by the computer compiling our code – it ignores them.

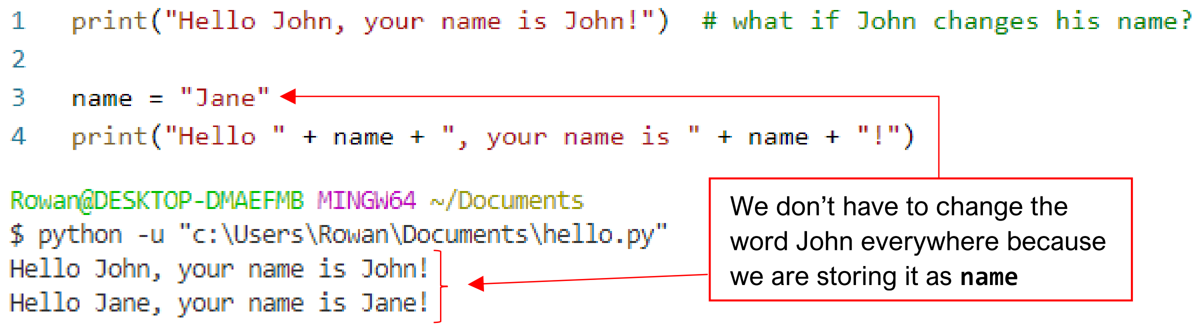
If we need to comment a paragraph or multiple lines, we can use **`“””comment“””.`**



```python
"""The code below prints
Hello World to the terminal"""
print("Hello World") #This prints Hello World to the terminal
```

Variables

Fine, so we can say words, but for us to do anything meaningful or usable in Python we’ll need to be able to store things that can vary. For example, given a username we can print it, but what if they change it? It is better if we store it and that way anytime they change it we can update the storage and everywhere it’s used will be updated. We call this storage type a **variable** – because it can vary!



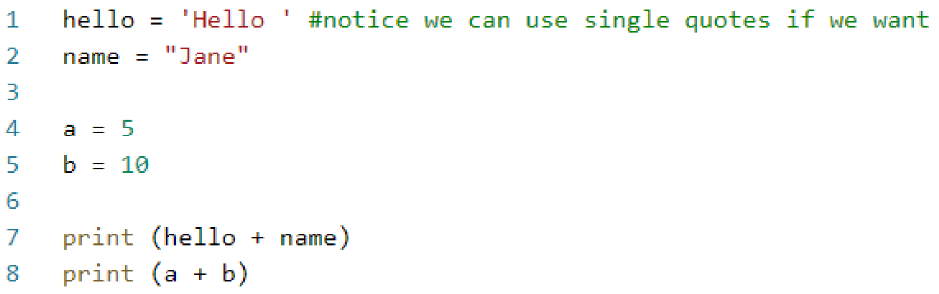
```python
print("Hello John, your name is John!")  # what if John changes his name?

name = "Jane"
print("Hello " + name + ", your name is " + name + "!")
```

```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents
$ python -u "c:\Users\Rowan\Documents\hello.py"
Hello John, your name is John!
Hello Jane, your name is Jane!
```

We don’t have to change the word John everywhere because we are storing it as **name**

The type of variable doesn’t have to be said (word/number), in fact they can change during the program. We can decide that **`name = 123`** and it would still print without errors!

You can use **+** to combine variables when printing, this works with words and numbers…



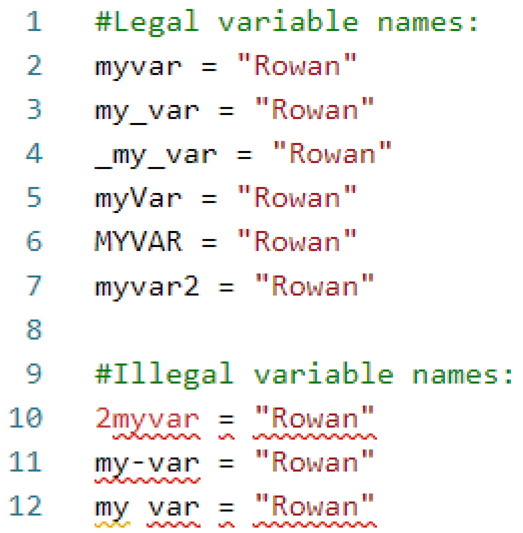
```python
hello = 'Hello ' #notice we can use single quotes if we want
name = "Jane"

a = 5
b = 10

print (hello + name)
print (a + b)
```

Saying some name **=** some data is called **variable assignment.** You are assigning that value to that name. You can assign multiple variables the same value `a = b = c = 100`

Or assign multiple variables in one line `x, y, z = 50, 100, 35`

We do have conventions (common rules) to follow when naming variables:

- Must start with a letter or the underscore character
- Cannot start with a number
- Can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Case-sensitive (age, Age and AGE are three different variables)

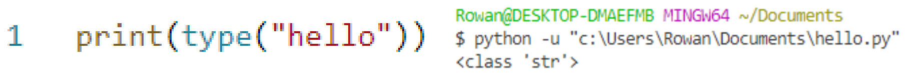


```
#Legal variable names:
myvar = "Rowan"
my_var = "Rowan"
_my_var = "Rowan"
myVar = "Rowan"
MYVAR = "Rowan"
myvar2 = "Rowan"

#Illegal variable names:
2myvar = "Rowan"
my-var = "Rowan"
my var = "Rowan"
```

Data Types

We've touched on this idea of words/numbers but what really is a word? Well programming languages have specific types of data so that we understand exactly what everything is.

To find the type of anything we can use the **type()** function.



```
print(type("hello"))
```

```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents
$ python -u "c:\Users\Rowan\Documents\hello.py"
<class 'str'>
```

We'll look at the following built-in data types (however there are more!):

Name: String **Symbol: `str`** `name = "Rowan"`

A word is just some characters *stringed* together. So, anything involving text characters is referred to as a string – in Python we use **`str`.** (Above "**`hello`**" is of the type/class "**`str`**")

Name: Integer **Symbol: `int`** `age = 35`

A mathematical integer is any *whole* number (no fractional part). In Python **`int`**.

Name: Float **Symbol: `float`** `pi = 3.14159265359`

A floating-point number is any number with a *decimal point* (fractional part). In Python **`float`**.

Name: Boolean **Symbol: `bool`** `is_light_on = True`

In the intro I mentioned computers only deal in 0s and 1s – we can interpret this as something is either true or false. Booleans represent this, a Boolean type variable can either be **`True`** or **`False`.** In Python **`bool`**.

Name: List **Symbol: `list`** `fruits = ["apple", "banana", "orange"]`

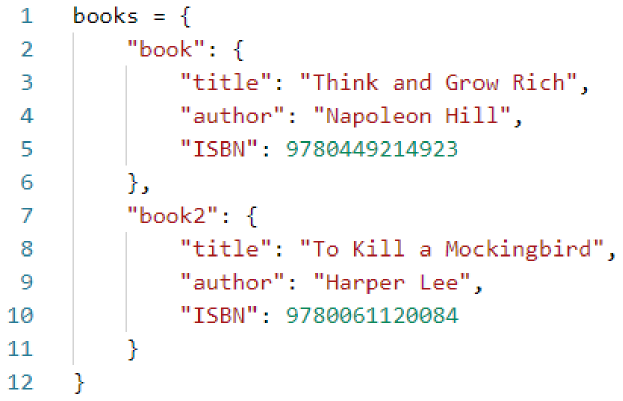
A list is a collection of data (any types) that is **ordered** and **changeable** and can include duplicates of the same data. In Python **`list`**.

Name: Tuple **Symbol: `tuple`** `fruits = ("apple", "banana", "orange")`

A tuple is a collection of data (any types) that is **ordered** and **unchangeable** but still can include duplicates of the same data. In Python **`tuple`**.

Name: Dictionary **Symbol: `dict`**

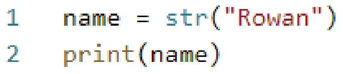
A dictionary is a collection of data (any types) that is **unordered** and **changeable** but cannot have duplicate data and it indexes the data. So, we can say – give me the data at this location, like how we can go to the "A" section of the Oxford dictionary and look for the word "Apple" and the data for its definition can be found. This is indexing. In Python **`dict`**.



```
books = {
    "book": {
        "title": "Think and Grow Rich",
        "author": "Napoleon Hill",
        "ISBN": 9780449214923
    },
    "book2": {
        "title": "To Kill a Mockingbird",
        "author": "Harper Lee",
        "ISBN": 9780061120084
    }
}
```

Type Declaration

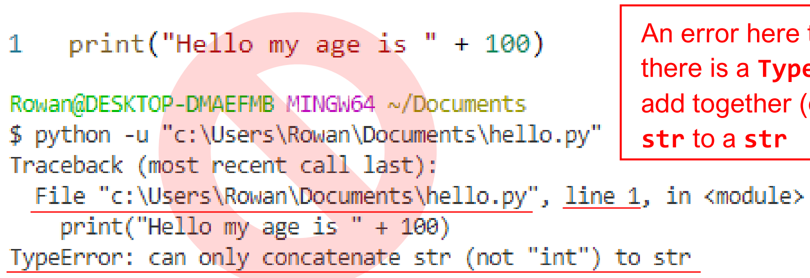
Python doesn’t care what type a variable is, but if you want to enforce a certain type, each type has a **constructor function** used in the following way…



```python
name = str("Rowan")
print(name)
```

Type Casting (Type Conversion)

Sometimes we need to treat types as something else to use it – for example we want to print a string and a number.

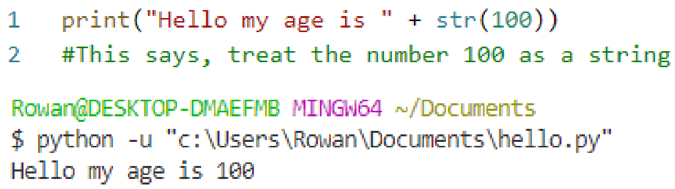


```python
1   print("Hello my age is " + 100)
```

```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents
$ python -u "c:\Users\Rowan\Documents\hello.py"
Traceback (most recent call last):
  File "c:\Users\Rowan\Documents\hello.py", line 1, in <module>
    print("Hello my age is " + 100)
TypeError: can only concatenate str (not "int") to str
```

An error here tells us on line 1 there is a **`TypeError`** it can only add together (concatenate) a **`str`** to a **`str`**

We can **cast** our number to a string, this is called **`type casting`** and works as follows…



```python
print("Hello my age is " + str(100))
#This says, treat the number 100 as a string
```

```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents
$ python -u "c:\Users\Rowan\Documents\hello.py"
Hello my age is 100
```

See, it is like we’re constructing a string with the number 100, similar to how we declared data type above by constructing!

*string/integer to a f*loat using **`float().`**

string/float to an integer using **`int().`**

integer/float to a string using **`str().`**

To cast strings, they must clearly be numbers - **“`54`”** can be cast, but **“`hello`”** cannot.

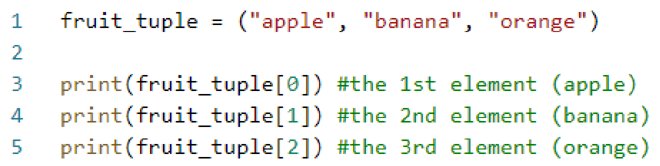
Collections

Suppose we have many variables, or many pieces of data and we want to store them? We'd put them in a filing cabinet, or a drawer (in the real world)! In Python we touched on several collection data types earlier – now let's go into more detail…

`list` and `tuple`

We can access the elements (piece of data) inside a list or a tuple using its **index.** However, whilst we naturally think the 1st thing in the collection will be at index 1 and 2nd at index 2 etc. ***this is not the case in programming***! Instead we **count from 0**. So the 1st element in the list or tuple is **0,1,2,3,etc.**

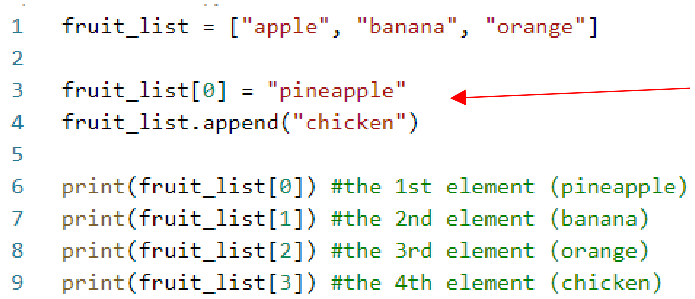
Knowing this we can access the element we want in the collection using `[]` to represent the index we want to look inside – think of the index as like a sign on a bucket, where each bucket has something inside it:



```python
fruit_tuple = ("apple", "banana", "orange")

print(fruit_tuple[0]) #the 1st element (apple)
print(fruit_tuple[1]) #the 2nd element (banana)
print(fruit_tuple[2]) #the 3rd element (orange)
```

Recall that the tuple data type is **unchangeable** so we cannot add/update once we have assigned it. It remains exactly as you declared it.

Lists are still unordered, but are changeable! We can change the data at each index (like changing what is inside the bucket) and even add more buckets with more data! Remember declaring a list needs `[]` brackets instead of `()` brackets.



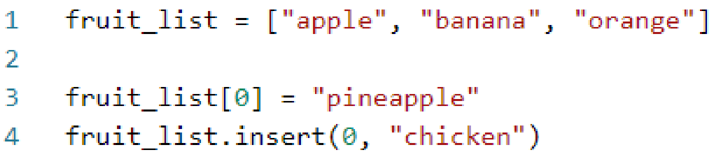
```python
fruit_list = ["apple", "banana", "orange"]

fruit_list[0] = "pineapple"
fruit_list.append("chicken")

print(fruit_list[0]) #the 1st element (pineapple)
print(fruit_list[1]) #the 2nd element (banana)
print(fruit_list[2]) #the 3rd element (orange)
print(fruit_list[3]) #the 4th element (chicken)
```

See how we updated the data at the first index and that we also added more data to the list!

To add more data to a list we **append** to it. Some data types in Python have "functions" they can perform – they typically take some input (known as an argument). Here we are saying let the **`fruit_list`** variable use its **append** function on the word **"`chicken`"** to append a new value to itself. Notice we can now print the 4th element.

To call a function on a variable in python we use "." (dot) which composes them together. It says the function after the "." (dot) belongs to the item before the "." (dot).

To add an item to a specific index (and move everyone else over by 1 space) we can use the **insert** function.

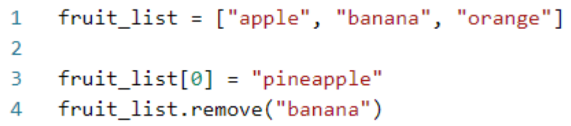


```python
fruit_list = ["apple", "banana", "orange"]

fruit_list[0] = "pineapple"
fruit_list.insert(0, "chicken")
```

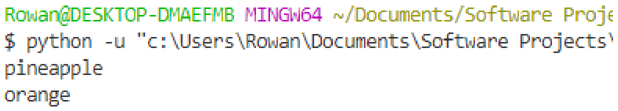
This makes element 0 now **"`chicken`"** and **`pineapple` becomes index 1** (shifted right)

We can also **remove** items from lists:

Using **`remove()`** function lets us remove data –

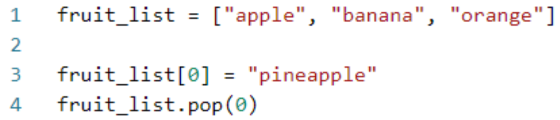


```
fruit_list = ["apple", "banana", "orange"]

fruit_list[0] = "pineapple"
fruit_list.remove("banana")
```

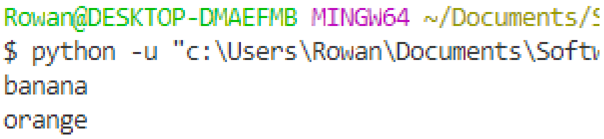


```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software Proje
$ python -u "c:\Users\Rowan\Documents\Software Projects\
pineapple
orange
```

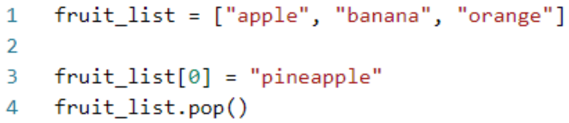
Using **`pop()`** removes data at a given index (no input to the function removes final element) –



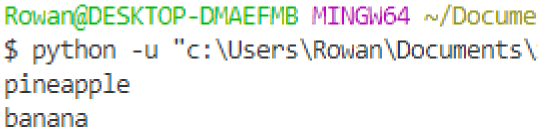
```
fruit_list = ["apple", "banana", "orange"]

fruit_list[0] = "pineapple"
fruit_list.pop(0)
```



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/S
$ python -u "c:\Users\Rowan\Documents\Softw
banana
orange
```

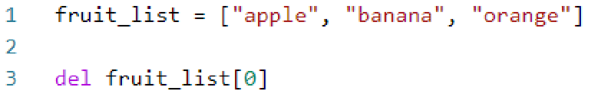


```
fruit_list = ["apple", "banana", "orange"]

fruit_list[0] = "pineapple"
fruit_list.pop()
```

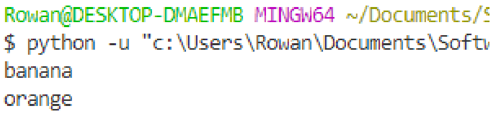


```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Docume
$ python -u "c:\Users\Rowan\Documents\
pineapple
banana
```

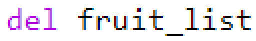
Using the **`del`** keyword we can remove a specific index or the list in entirety –



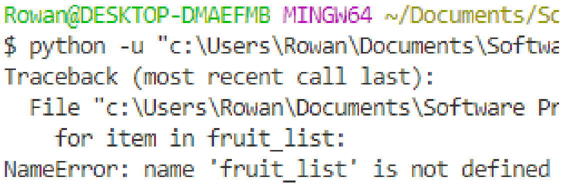
```
fruit_list = ["apple", "banana", "orange"]

del fruit_list[0]
```



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/S
$ python -u "c:\Users\Rowan\Documents\Softw
banana
orange
```

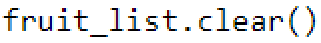


```
del fruit_list
```



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Sc
$ python -u "c:\Users\Rowan\Documents\Softwa
Traceback (most recent call last):
  File "c:\Users\Rowan\Documents\Software Pr
    for item in fruit_list:
NameError: name 'fruit_list' is not defined
```

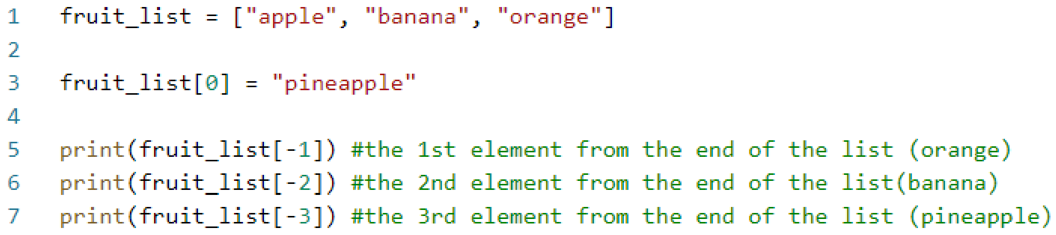
Using **`clear()`** we can empty a list – this means the list remains defined, just empty.



```
fruit_list.clear()
```

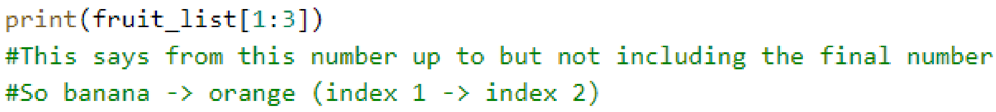
We can also combine lists using the **+** notation as we did with other data types :)

Further to using regular index numbers we can also use **negative indexes** to count from the end of the list.

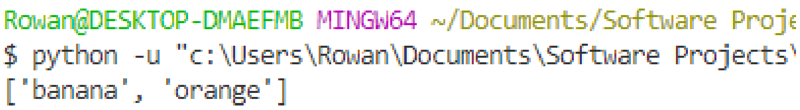


```
fruit_list = ["apple", "banana", "orange"]

fruit_list[0] = "pineapple"

print(fruit_list[-1]) #the 1st element from the end of the list (orange)
print(fruit_list[-2]) #the 2nd element from the end of the list(banana)
print(fruit_list[-3]) #the 3rd element from the end of the list (pineapple)
```

We can even use access a range of indexes if we only want specific parts:

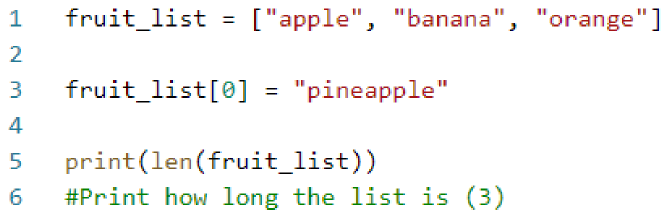


```
print(fruit_list[1:3])
#This says from this number up to but not including the final number
#So banana -> orange (index 1 -> index 2)
```



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software Proje
$ python -u "c:\Users\Rowan\Documents\Software Projects\
['banana', 'orange']
```

To find the length of elements in a list we can use the **`len()`** function similar to the use of the **`type()`** function to find the type of a variable.



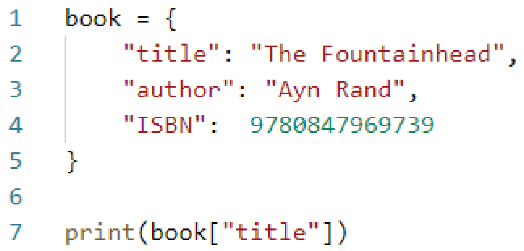
```
fruit_list = ["apple", "banana", "orange"]

fruit_list[0] = "pineapple"

print(len(fruit_list))
#Print how long the list is (3)
```

This output is “3”

Always remember – you CANNOT update/alter/remove a tuple after its creation! Only lists!

dict

Dictionaries are **unordered, changeable,** and **indexed.** The data types section covered how we define a series of “keys” and their values. Similar to how we define words and their definitions in a traditional dictionary. To get the value for a given “key” we use **`[]` notation** however **we use the key title**, *not an index number*.



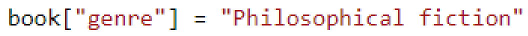
```
book = {
    "title": "The Fountainhead",
    "author": "Ayn Rand",
    "ISBN":  9780847969739
}

print(book["title"])
```

We could also use **`book.get(“title”)`** and use the dictionary variable’s **`get()`** function for any of its keys.

You can update a key’s value the same as you would a variable by simply reassigning it using a =

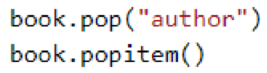
You can find how many items (key-value pairs) are in a dictionary using the **`len()`** function similar to the other collection types.

To add a new pair simply assign a value to some new key name.



```
book["genre"] = "Philosophical fiction"
```

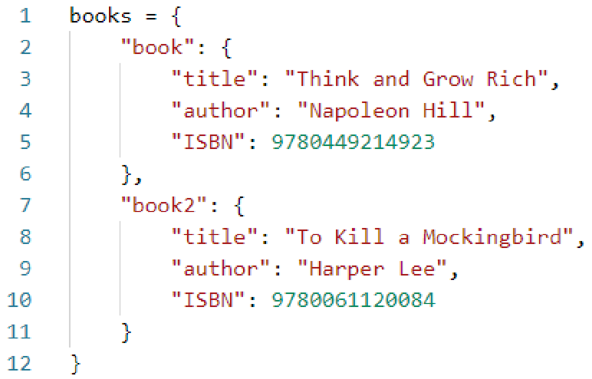
To remove a pair you can use the **`pop()`** function just like the other collection types, **however if you want to remove the most recent item you must use `popitem()`**



```
book.pop("author")
book.popitem()
```

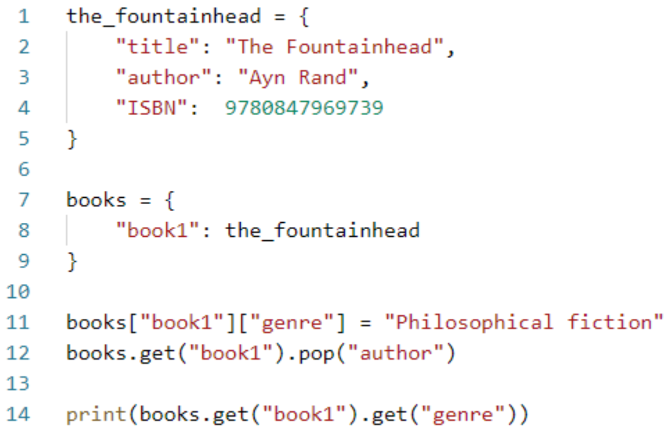
Dictionaries also allow the use of the **del** keyword and **clear()** function!

You may have noticed the example of books earlier was multiple “layers” deep. You can put dictionaries inside dictionaries – here it is again, notice multiple dictionaries are **nested** inside one another.



```
books = {
    "book": {
        "title": "Think and Grow Rich",
        "author": "Napoleon Hill",
        "ISBN": 9780449214923
    },
    "book2": {
        "title": "To Kill a Mockingbird",
        "author": "Harper Lee",
        "ISBN": 9780061120084
    }
}
```

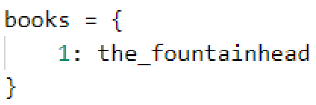
You can decide to nest dictionaries by using dictionary variables in the values instead of declaring them inside one another



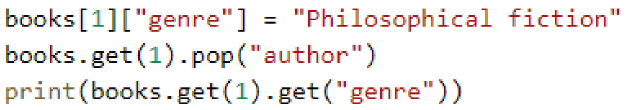
```python
the_fountainhead = {
    "title": "The Fountainhead",
    "author": "Ayn Rand",
    "ISBN":  9780847969739
}

books = {
    "book1": the_fountainhead
}

books["book1"]["genre"] = "Philosophical fiction"
books.get("book1").pop("author")

print(books.get("book1").get("genre"))
```

Take a minute to understand and use this example. Notice how we have had to change the way we're accessing because now we need to get the value of a value (it is 2 dictionaries deep).

Note that *a dictionary's keys can be numbers* just as the values can. Meaning this is valid:



```python
books = {
    1: the_fountainhead
}
```



```python
books[1]["genre"] = "Philosophical fiction"
books.get(1).pop("author")
print(books.get(1).get("genre"))
```

Conditionals

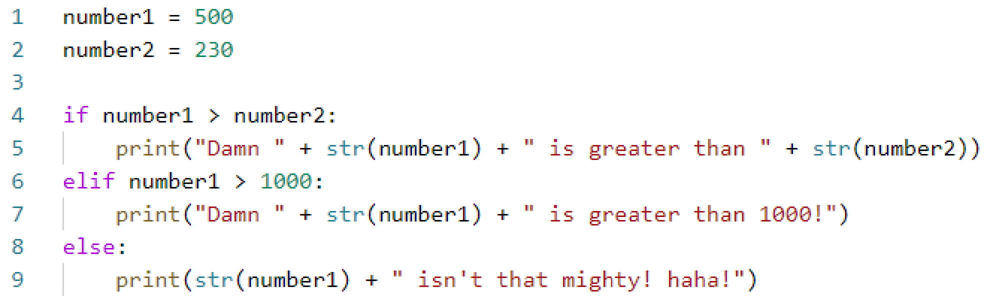
These are a core part of life – and computer science. We often find ourselves saying things such as **if it rains today, I'll get wet** and can go into further detail and say **also if it rains today and I have an umbrella, I won't get wet**. These are known as conditional sentences[vii].

It's super useful that we can make these kinds of statements in our code as well! Python uses the following structure for this – **`if -> elif -> else`.** It checks the first condition (is it true), if not then it checks the "elif" (meaning "or else if") and we can have any number of these elifs, before finally providing an "else" if none of the prior conditions were met. The else is a catch all keyword – "anything else: do this".

It's important to know how we can compare things in Python in order to be able to give good conditions for our statements. Python uses the normal mathematical conditions:

- Equals: **`a == b`**
- Not Equals: **`a != b`**
- Less/greater than: **`a < b`** or **`a > b`**
- Less/greater than or equal to: **`a <= b`** or **`a >= b`**

So now we can do something like this –

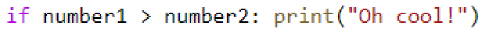


```python
number1 = 500
number2 = 230

if number1 > number2:
    print("Damn " + str(number1) + " is greater than " + str(number2))
elif number1 > 1000:
    print("Damn " + str(number1) + " is greater than 1000!")
else:
    print(str(number1) + " isn't that mighty! haha!")
```

Try experimenting with changing **`number1`** and **`number2`** values and even add your own conditions by adding more **`elif`** clauses!

Indenting by pressing the "Tab" key is essential here (to push the code forward)! It's also essential for any nested structure in our code. Pressing "Shift + Tab" will bring the indented code back!

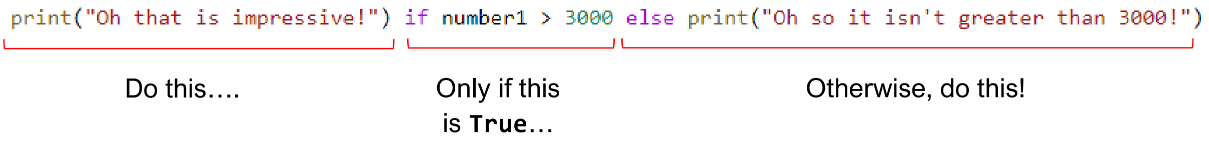
If you only need 1 "if" clause you can make your conditional 1 line:



```python
if number1 > number2: print("Oh cool!")
```

Bonus: ***Ternary Conditional – this lets us say "if, else" statements in 1 line!***

Simply write your statement as below to use this style of conditional (useful in "either this condition is met or anything else" scenarios…



```python
print("Oh that is impressive!") if number1 > 3000 else print("Oh so it isn't greater than 3000!")
```

Do this….

Only if this is **`True`**…

Otherwise, do this!

But we can do even more with conditional statements…

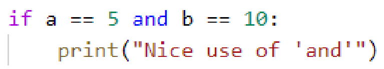
Operators

Logical operators

Ok how about we wanted to make sure 2 statements are True for something to happen, “if the weather is sunny and I don’t have sunglasses then I will buy sunglasses”. We have several scenarios where logic like this is involved. Below are some logical operators (the name for words that deal with logic) that can be used in Python…

`and`

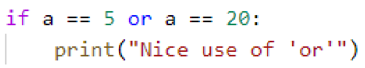
Ensure 2 conditions are true…



```
if a == 5 and b == 10:
    print("Nice use of 'and'")
```

`or`

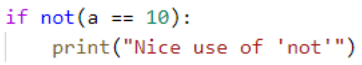
Ensure either or both are true (at least 1 must be true)…



```
if a == 5 or a == 20:
    print("Nice use of 'or'")
```

`not`

Reverse the result of a condition – if it is true you are now saying it must be false…



```
if not(a == 10):
    print("Nice use of 'not'")
```

“a is **not** equal to 10” is the same as thinking “a is equal to 10” is false.

Arithmetic operators

We’ve seen many of these already but here are all the mathematical arithmetic operators you can use in Python…

- `+` (Addition) – `5+10 = 15`
- `-` (Subtraction) – `10-7 = 3`
- `*` (Multiplication) – `5*5 = 25`
- `/` (Division) – `6/4 = 1.5`
- `%` (Modulus) – take the remainder of a division. No remainder = 0. So `15%4 = 3.`
- `**` (Exponentiation) – for example, 2^3 would be `2**3.`
- `//` (Floor division) – this takes the closest smaller number for example the result of a division being 1.9 becomes 1. For example, `6//4 = 1.` (It takes the floor!)

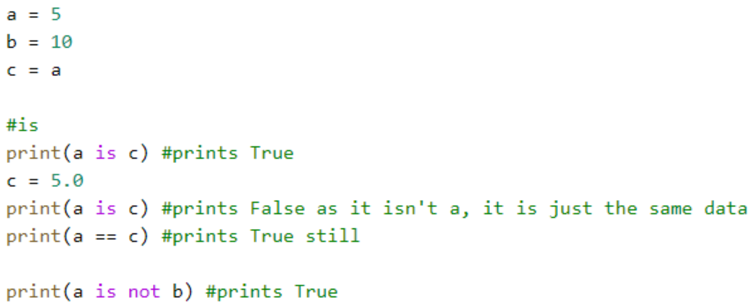
Identity operators

`is`

True if both variables are the same (not the same content but exactly the same object)

`is not`

True if both variables are not the same object.



```
a = 5
b = 10
c = a

#is
print(a is c) #prints True
c = 5.0
print(a is c) #prints False as it isn't a, it is just the same data
print(a == c) #prints True still

print(a is not b) #prints True
```

Assignment operators

There can be more useful than just "=" for variable assignment:

Operator	Example	Same As
`+=`	`x += 1`	`x = x + 1`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 2`	`x = x * 2`
`/=`	`x /= 3`	`x = x / 3`
`%=`	`x %= 2`	`x = x % 2`
`//=`	`x //= 4`	`x = x // 4`
`**=`	`x **= 2`	`x = x ** 2`

Other assignment operators exist but they won't be needed for this course.

Membership operators

`in`

True if a collection contains the same value as the other variable. For example…

```
fruits_tuple = ("orange", "banana", "kiwi")
fruit = "banana"

print(fruit in fruits_tuple) #prints True
```

`not in`

True if a collection does not contain a value of some variable…

```
print(fruit not in fruits_tuple) #prints False because it is present
```

Other operators exist but we won't need them for this course – although you can look into them to develop your programming and Python understanding further![viii]

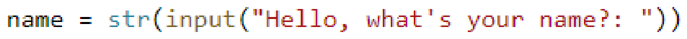
Exercise A – Fruit Chat

For each exercise you will start with barebones (skeleton) code to work on :)

Now we know most of the basic syntax we'll need, and we understand data types! Let's develop our first program…

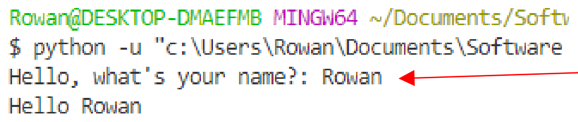
To take input from the user (person on the computer) we use the `input` keyword.

We cast the input to the form we want to use it as. In this exercise I've casted name to a String.



```
name = str(input("Hello, what's your name?: "))
```

Task: Output a "hello" message using their name. Your output should look similar to this…

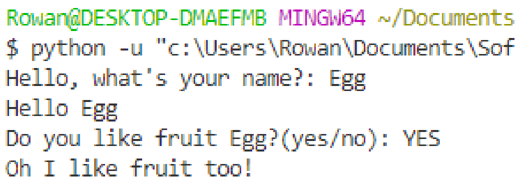


```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Softw
$ python -u "c:\Users\Rowan\Documents\Software
Hello, what's your name?: Rowan
Hello Rowan
```

To input, type in the terminal!

Task: Now ask the user if they like fruit – store their answer in a variable and use a conditional clause to check their answer. If they say yes perhaps say something like "Oh, cool I like fruit too!"

Pro-tip: ***we can use* `str.lower(input(…))` *to make any input lowercase. So even if they write "YES", our variable will store "yes". There is also a* `str.upper(input(…))`**

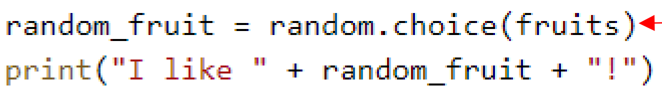


```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents
$ python -u "c:\Users\Rowan\Documents\Sof
Hello, what's your name?: Egg
Hello Egg
Do you like fruit Egg?(yes/no): YES
Oh I like fruit too!
```

Are you starting to see why this exercise is called "Fruit Chat" ;-)

Sometimes other people have written very useful pieces of code for the community to use instead of us having to rewrite it in every program. This code is held together in "packages". Python comes with some packages built in. One of these is called **`random`**. It deals with anything to do with generating something random. You may have noticed **to import a package in Python we use the `import` keyword. Hence your skeleton code contains `import random`.**

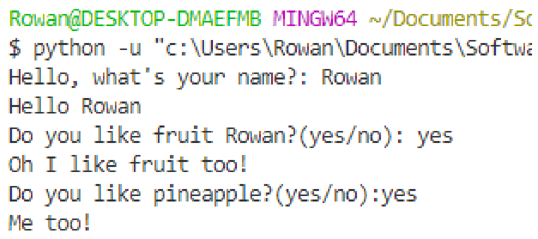
Below is an example of how we can use random to get a random element from our fruit list:



```
random_fruit = random.choice(fruits)
print("I like " + random_fruit + "!")
```

Take a random choice from this collection and store it in this variable

Task: If the user says the like fruit, ask them if they like some random fruit from our fruits list and say something in reply based on their answer.



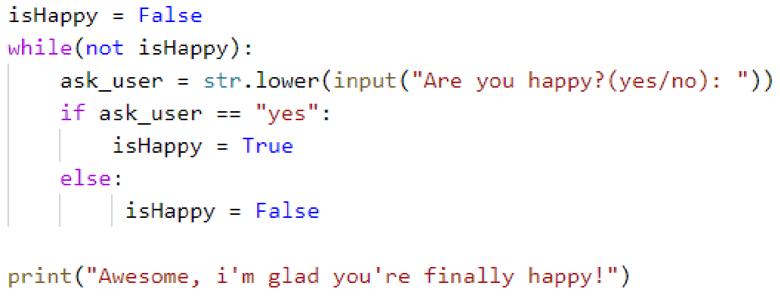
```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Sc
$ python -u "c:\Users\Rowan\Documents\Softwa
Hello, what's your name?: Rowan
Hello Rowan
Do you like fruit Rowan?(yes/no): yes
Oh I like fruit too!
Do you like pineapple?(yes/no):yes
Me too!
```

Loops

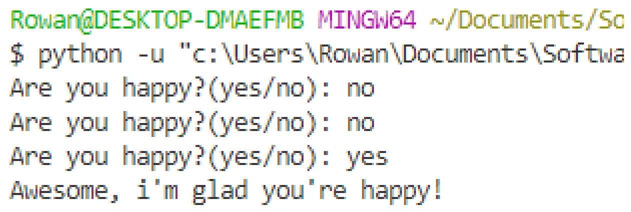
Sometimes we wish to repeat the same thing several times. We may want to repeat infinitely, or until a condition is met or just a set number of times. Fortunately, we can do all 3 of these in code! This is useful in asking for repeated input – consider when you enter a password wrong, you are typically asked to try again and following 3 retries you may be locked out for a given amount of time. There is many other scenarios where we'll need to repeat the same thing or keep taking input.

while

Named because the code indented in these loops will be executed "while this condition is true". Upon the condition becoming false the loop will finish.



```
isHappy = False
while(not isHappy):
    ask_user = str.lower(input("Are you happy?(yes/no): "))
    if ask_user == "yes":
        isHappy = True
    else:
        isHappy = False

print("Awesome, i'm glad you're finally happy!")
```



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Sc
$ python -u "c:\Users\Rowan\Documents\Softwa
Are you happy?(yes/no): no
Are you happy?(yes/no): no
Are you happy?(yes/no): yes
Awesome, i'm glad you're happy!
```

not isHappy is the same as **isHappy == False** as the default value would be **True**

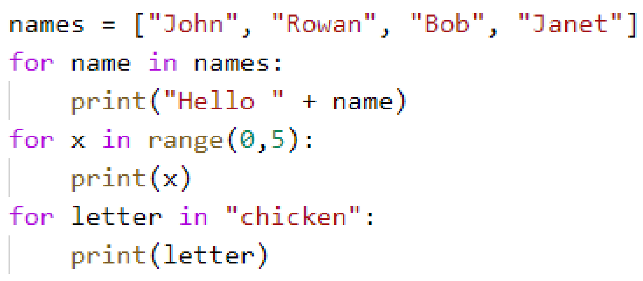
It will continue assigning the **ask_user** variable and going through the conditional until **isHappy == True**

Notice the input must be "yes" for this Boolean **isHappy** to change.

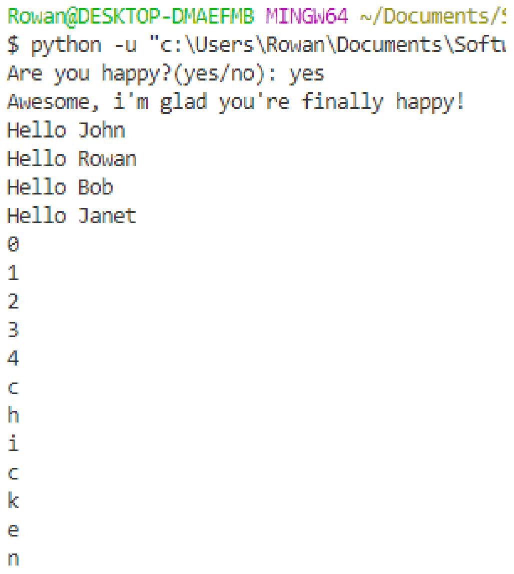
You need to include some code that will stop the while loop by making its condition false.

for

Named because the code will be executed "for some given sequence". This type of loop works like an inspector (iterator) which can go through each item in a given sequence – meaning it loops as many times as there is something in the sequence.



```
names = ["John", "Rowan", "Bob", "Janet"]
for name in names:
    print("Hello " + name)
for x in range(0,5):
    print(x)
for letter in "chicken":
    print(letter)
```

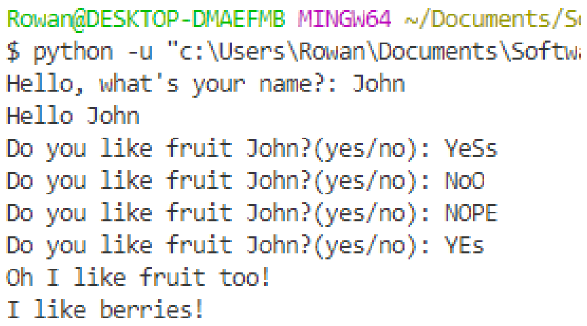


```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/S
$ python -u "c:\Users\Rowan\Documents\Softu
Are you happy?(yes/no): yes
Awesome, i'm glad you're finally happy!
Hello John
Hello Rowan
Hello Bob
Hello Janet
0
1
2
3
4
c
h
i
c
k
e
n
```

We are creating a temporary variable when we say for "something". This could be anything but typically it should be relevant – hence I used **for name** when iterating through the names in the list.

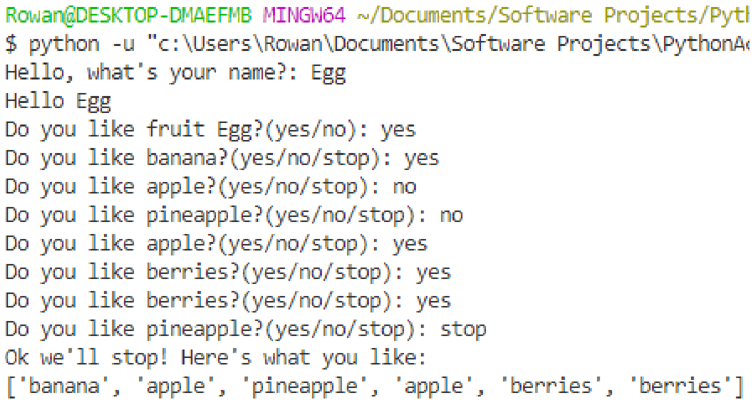
Exercise A – Fruit Chat (Extended)

Task: Now we know how loops work, add a loop to make sure the user cannot continue until they enter yes **or** no.



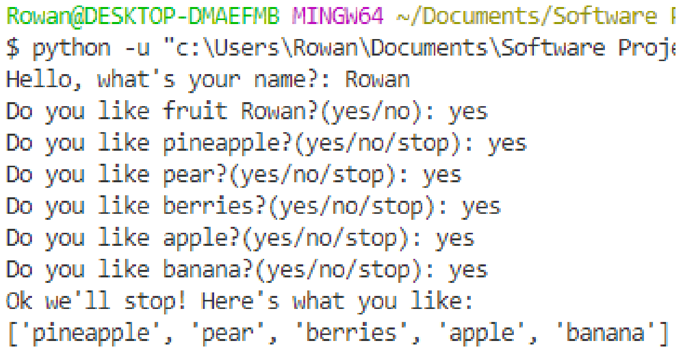
```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/S
$ python -u "c:\Users\Rowan\Documents\Softw
Hello, what's your name?: John
Hello John
Do you like fruit John?(yes/no): YeSs
Do you like fruit John?(yes/no): NoO
Do you like fruit John?(yes/no): NOPE
Do you like fruit John?(yes/no): YEs
Oh I like fruit too!
I like berries!
```

Novice Challenge (Optional): After the user has said they like fruit, ask if they like a random fruit – if they do, add it to a list. If at any time they say **stop**, then quit asking and show them the fruits they liked.



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software Projects/Pyt
$ python -u "c:\Users\Rowan\Documents\Software Projects\PythonA
Hello, what's your name?: Egg
Hello Egg
Do you like fruit Egg?(yes/no): yes
Do you like banana?(yes/no/stop): yes
Do you like apple?(yes/no/stop): no
Do you like pineapple?(yes/no/stop): no
Do you like apple?(yes/no/stop): yes
Do you like berries?(yes/no/stop): yes
Do you like berries?(yes/no/stop): yes
Do you like pineapple?(yes/no/stop): stop
Ok we'll stop! Here's what you like:
['banana', 'apple', 'pineapple', 'apple', 'berries', 'berries']
```

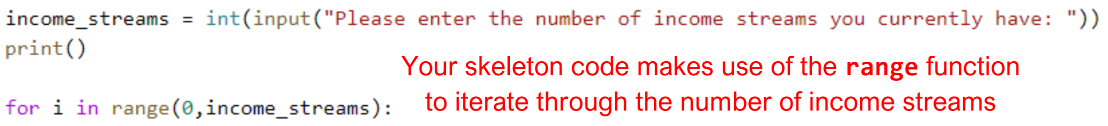
Expert Challenge (Optional): As a super far challenge (don't worry if you cannot) then make sure they are asked random fruit, but they are never asked about the same fruit twice – that is, there should be no duplicates in the fruits they like.



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software P
$ python -u "c:\Users\Rowan\Documents\Software Proj
Hello, what's your name?: Rowan
Do you like fruit Rowan?(yes/no): yes
Do you like pineapple?(yes/no/stop): yes
Do you like pear?(yes/no/stop): yes
Do you like berries?(yes/no/stop): yes
Do you like apple?(yes/no/stop): yes
Do you like banana?(yes/no/stop): yes
Ok we'll stop! Here's what you like:
['pineapple', 'pear', 'berries', 'apple', 'banana']
```

Exercise B – Gross Income Calculator

Using our fundamentals, we can now implement something useful. A simple calculator to find your current annual take home pay:

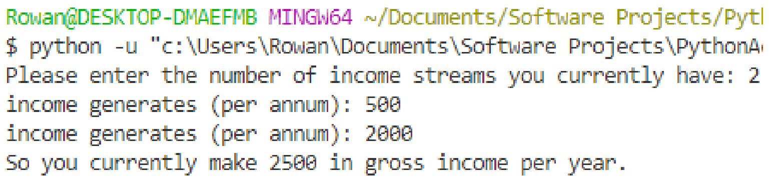


```
income_streams = int(input("Please enter the number of income streams you currently have: "))
print()

for i in range(0,income_streams):
```

Your skeleton code makes use of the **range** function to iterate through the number of income streams

Task: Take a number of income streams from the user, then using a loop to go through them find the total amount of money they earn per year before any deductions (sum of each income). Finally print the total amount to the user.

Pro-tip: ***it may help to have a variable which begins at 0 and grows with each input taken from the user***

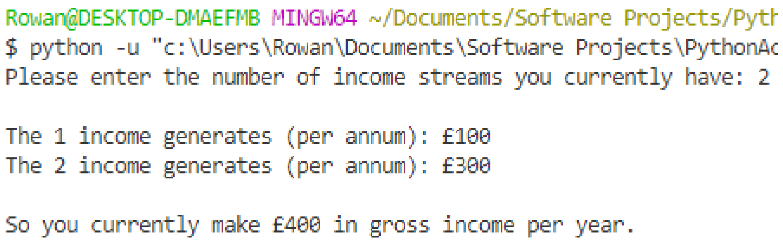


```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software Projects/Pytl
$ python -u "c:\Users\Rowan\Documents\Software Projects\PythonA
Please enter the number of income streams you currently have: 2
income generates (per annum): 500
income generates (per annum): 2000
So you currently make 2500 in gross income per year.
```

Novice Challenge (Optional): Improve the visual look of your output, make use of the £ or $ sign and show a clear difference between questions asked.

Pro-tip: ***consider using the iterator variable in your loop to keep track of which income stream your asking about (1,2,3, etc.)***

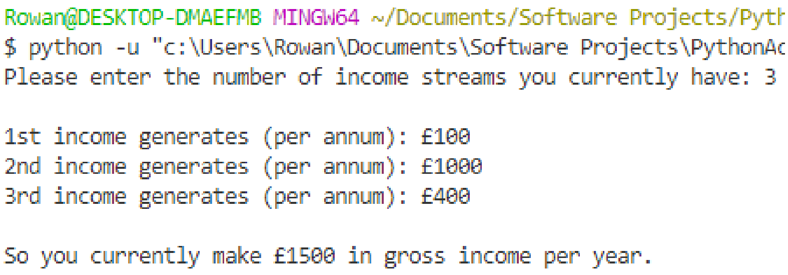
Pro-tip: ***Using "`\n`" in a string creates a new line – think how it can be used here in your outputs.***



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software Projects/Pytl
$ python -u "c:\Users\Rowan\Documents\Software Projects\PythonAc
Please enter the number of income streams you currently have: 2

The 1 income generates (per annum): £100
The 2 income generates (per annum): £300

So you currently make £400 in gross income per year.
```

Expert Challenge (Optional): Add suffixes to each income stream. So "st", "nd", "rd" and further will be "th".

Pro-tip: ***Use a dictionary and it's `get()` function to map your iterator to the correct suffix. The `get()` function has a default option if it cannot find the item!***

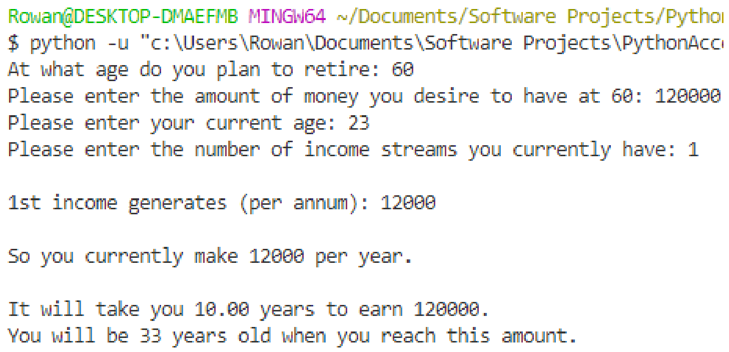


```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software Projects/Pytl
$ python -u "c:\Users\Rowan\Documents\Software Projects\PythonAc
Please enter the number of income streams you currently have: 3

1st income generates (per annum): £100
2nd income generates (per annum): £1000
3rd income generates (per annum): £400

So you currently make £1500 in gross income per year.
```

Exercise C – Retirement

We're now going to look at how we can deal with income and time to gauge net worth at a given age. Our model (program) is very simple however you may very easily tweak it to achieve a realistic pension calculator. Vanguard have a perfect example of something easily achievable using this exercise[ix].

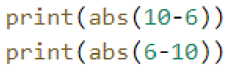
Task: Find the number of years it will take the user to achieve a desired income based on their current earnings, without accounting for growth or inflation.



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software Projects/Pytho
$ python -u "c:\Users\Rowan\Documents\Software Projects\PythonAcc
At what age do you plan to retire: 60
Please enter the amount of money you desire to have at 60: 120000
Please enter your current age: 23
Please enter the number of income streams you currently have: 1

1st income generates (per annum): 12000

So you currently make 12000 per year.

It will take you 10.00 years to earn 120000.
You will be 33 years old when you reach this amount.
```

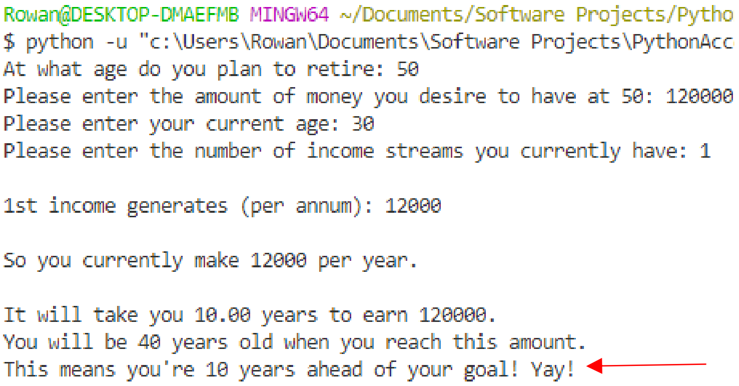
Novice Challenge (Optional): If they are on track to meet their desired net worth at this retirement age, show how many years they are ahead. If they are behind show how many years.

Pro-tip: ***the* `abs()` *function will calculate the absolute difference in a calculation***



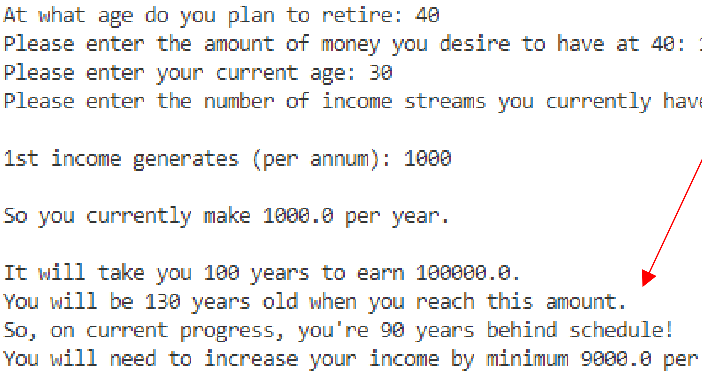
```
print(abs(10-6))
print(abs(6-10))
```

Both output 4!

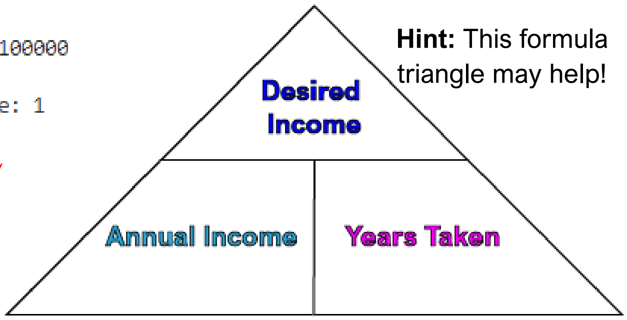


```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software Projects/Pytho
$ python -u "c:\Users\Rowan\Documents\Software Projects\PythonAcc
At what age do you plan to retire: 50
Please enter the amount of money you desire to have at 50: 120000
Please enter your current age: 30
Please enter the number of income streams you currently have: 1

1st income generates (per annum): 12000

So you currently make 12000 per year.

It will take you 10.00 years to earn 120000.
You will be 40 years old when you reach this amount.
This means you're 10 years ahead of your goal! Yay!
```

Expert Challenge (Optional): If behind schedule, show by how much they must increase their annual income to reach the desired income at the given retirement age.



```
At what age do you plan to retire: 40
Please enter the amount of money you desire to have at 40: 100000
Please enter your current age: 30
Please enter the number of income streams you currently have: 1

1st income generates (per annum): 1000

So you currently make 1000.0 per year.

It will take you 100 years to earn 100000.0.
You will be 130 years old when you reach this amount.
So, on current progress, you're 90 years behind schedule!
You will need to increase your income by minimum 9000.0 per year to reach your goal! Goodluck!
```

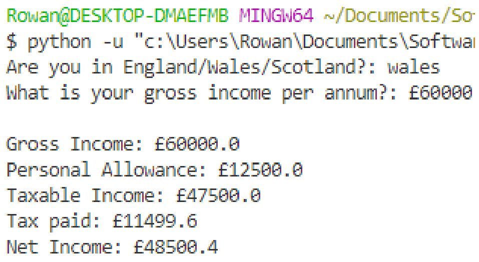


Hint: This formula triangle may help!

Exercise D – Net Income Calculator

Ok now we can adjust our thinking and account for tax and deductions. You can take this as far as you'd like – going further into national insurance, corporation tax, capital gains, and debt expenses etc. however **we will focus only on the UK income tax[x] bands.**

Task: This task is one of the most challenging but also very rewarding and can be extended later. Using the skeleton code take a users **gross income(pre-tax)** and using the **default personal allowance of £12500** with the provided **dictionary for tax bands and rates,** display to the user there take home income. ***Do not worry about National Insurance contributions :)***



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/So
$ python -u "c:\Users\Rowan\Documents\Softwa
Are you in England/Wales/Scotland?: wales
What is your gross income per annum?: £60000

Gross Income: £60000.0
Personal Allowance: £12500.0
Taxable Income: £47500.0
Tax paid: £11499.6
Net Income: £48500.4
```

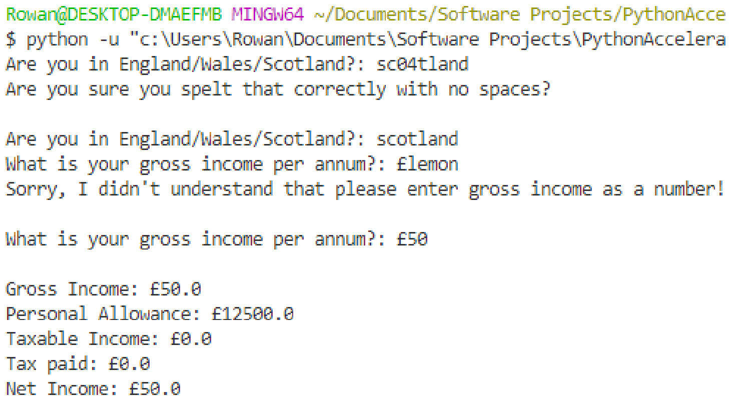
Pro-tip: ***Recall taxable income is their gross income minus personal allowance!***

Pro-tip: ***Each country has a list with each sub-list being a tax band. Position 0 is always the minimum for that band, 1 is the max for the band and 2 is the percentage (as a decimal)***

Pro-tip: ***The*** `max()` ***function returns the highest of given values –*** `max(5,200)` ***gives*** `200`

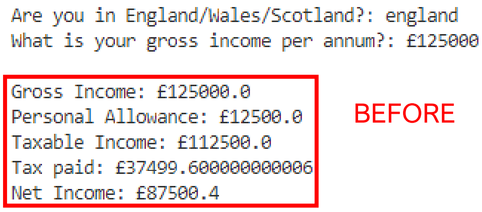
Pro-tip: ***Use a lists*** `sum()` ***function to sum up every value in a list –*** `sum([1,4])` ***gives*** `5`

Novice Challenge (Optional): Ensure only valid inputs let a user continue. For example entering a number when asked for country you will be repeatedly asked until one of the given options is given. Do this for both country and income.



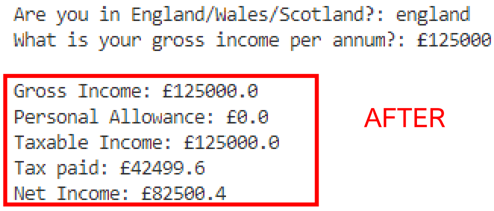
```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software Projects/PythonAcce
$ python -u "c:\Users\Rowan\Documents\Software Projects\PythonAccelera
Are you in England/Wales/Scotland?: sc04tland
Are you sure you spelt that correctly with no spaces?

Are you in England/Wales/Scotland?: scotland
What is your gross income per annum?: £lemon
Sorry, I didn't understand that please enter gross income as a number!

What is your gross income per annum?: £50

Gross Income: £50.0
Personal Allowance: £12500.0
Taxable Income: £0.0
Tax paid: £0.0
Net Income: £50.0
```

Expert Challenge (Optional): To make our model more accurate, after a user has entered their gross income – determine their personal allowance (instead of using £12500 for everyone). You need to check if they are earning £100,000+ and for every £2 over £100,000 their personal allowance should decrease by £1. This means an income of £125000 or higher has no personal allowance.



```
Are you in England/Wales/Scotland?: england
What is your gross income per annum?: £125000

Gross Income: £125000.0
Personal Allowance: £12500.0
Taxable Income: £112500.0
Tax paid: £37499.600000000006
Net Income: £87500.4
```

BEFORE



```
Are you in England/Wales/Scotland?: england
What is your gross income per annum?: £125000

Gross Income: £125000.0
Personal Allowance: £0.0
Taxable Income: £125000.0
Tax paid: £42499.6
Net Income: £82500.4
```

AFTER

Conclusion

Thank you so much for taking these steps into Python development. If you didn't spot it, I designed these exercises each with some area/use in mind for you to apply!

Through Exercise A you hopefully can begin to understand how AI works, in that it is taking a user's input, comparing it with the knowledge and statements it has gathered over time. This can give the illusion the computer has a mind, when in reality it is complex data and algorithms that allow it to "think" – much like how the program discussed fruit with the user and gave fitting answers!

Exercise B hopes to teach you the powers of Python in aiding daily and repetitive tasks. Sure we can sum and find our gross income ourselves using calculators or a notepad and pen, but through the use of Python and computers we can *write our own code* to help us with such tasks. The financial industry makes use of Python for all kinds of number crunching similar to this.

Exercise C aims to show you how given a set of inputs we can use computers to analyse and predict future outcomes. You can adapt this exercise to account for inflation or for some given growth factor and see how programs can be used to model future outcomes. Really nobody can predict the future – but using numbers and data we can make estimates!

Using Exercise D you can see how programs can be designed to be reusable and maintainable. If the tax bands or rates are changed, you can simply change their values and the program will still perform. We also can perform more complex calculations using programming – something that was hopefully clear in this exercise.

Having completed this course, you now have the underlying principles to understand Python. You can continue learning using https://www.w3schools.com/python/ where you can automatically skip to the *"functions"* section! Python really is one of the world's largest, most supported and most fun languages you can learn, and I strongly encourage you to keep completing increasingly difficult courses so that you may excel in this for years to come.

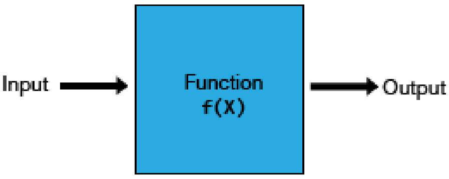
If you have any concerns regarding the course content itself, I would love to listen and address them at aldeansoftware@gmail.com.

~ Rowan Aldean

Bonus (Further Reading)

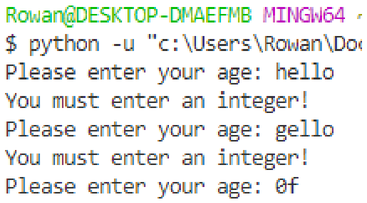
Functions

This course has looked at traditional procedural programming. The truth is, just like in math, there is often times where we want our code to perform a certain function. You can think of *squaring* in Math as a function. Give 2 to the "squared function" and 4 pops out! See?

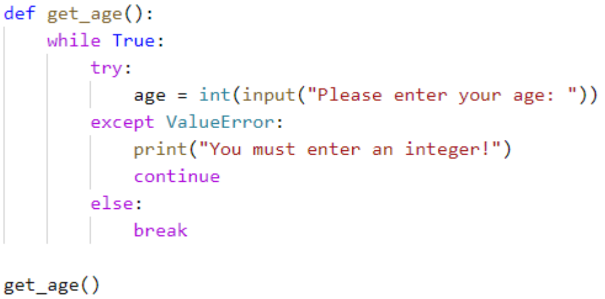


Addition can be seen as a function – **add(2,5)** it takes two numbers and adds them together to produce an output!

This is useful to separate the parts of our program – especially if it's a function we'll be using more than once! For example, in order to easily get input we may write a function like this:



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~
$ python -u "c:\Users\Rowan\Doc
Please enter your age: hello
You must enter an integer!
Please enter your age: gello
You must enter an integer!
Please enter your age: 0f
```



```
def get_age():
    while True:
        try:
            age = int(input("Please enter your age: "))
        except ValueError:
            print("You must enter an integer!")
            continue
        else:
            break


get_age()
```

Anywhere we wish to get age input we can now write **`get_age().`** This is how we "call" or execute a function. Without this the function is defined but it hasn't been used. We defined a function using the **`def`** keyword.

When using functions (and indented code) we have 2 types of variables.

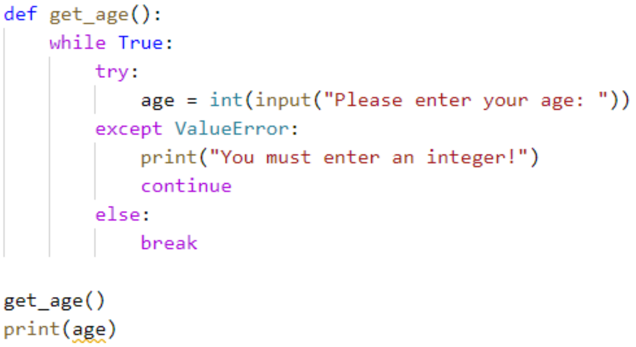
Local variables

These are declared inside an indented block or inside a function. Only code at that same level or below can use this variable. You cannot use it elsewhere or in another function.

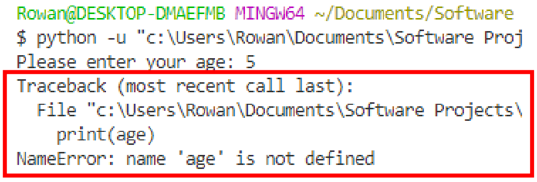
Global variables

These are variables which can be accessed anywhere in the program. We can change a local variable into a global one using the **global** keyword as follows…

We get an error trying to print a local variable

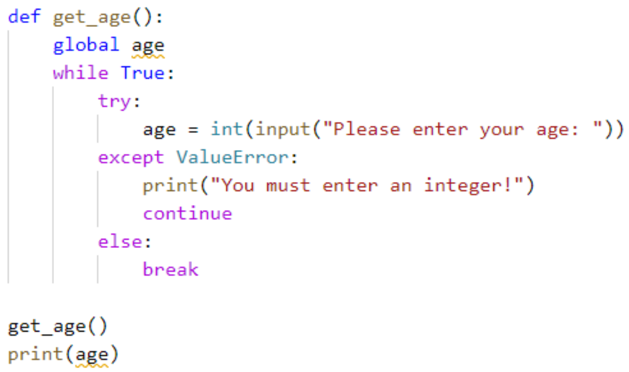


```
def get_age():
    while True:
        try:
            age = int(input("Please enter your age: "))
        except ValueError:
            print("You must enter an integer!")
            continue
        else:
            break


get_age()
print(age)
```

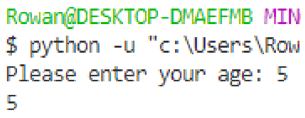


```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software
$ python -u "c:\Users\Rowan\Documents\Software Proj
Please enter your age: 5
Traceback (most recent call last):
  File "c:\Users\Rowan\Documents\Software Projects\
    print(age)
NameError: name 'age' is not defined
```

Declaring a global variable

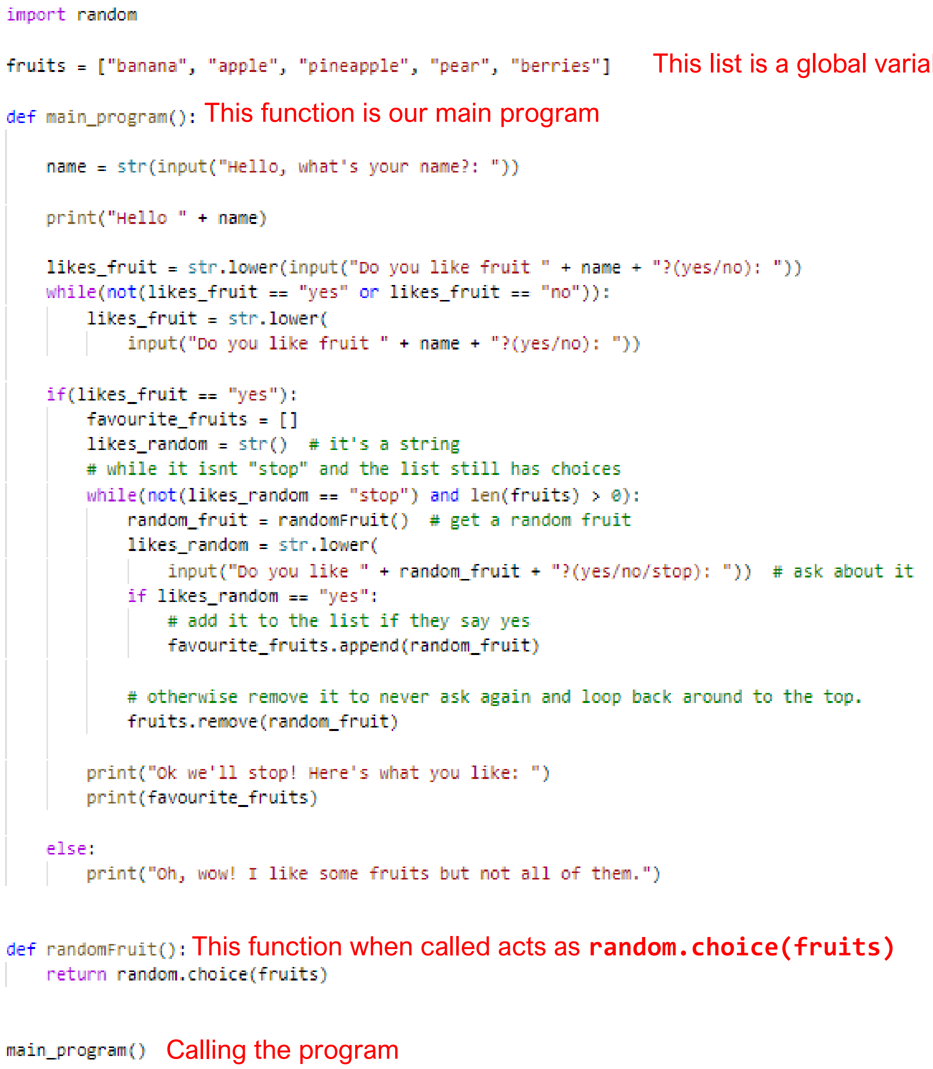


```
def get_age():
    global age
    while True:
        try:
            age = int(input("Please enter your age: "))
        except ValueError:
            print("You must enter an integer!")
            continue
        else:
            break

get_age()
print(age)
```



```
Rowan@DESKTOP-DMAEFMB MIN
$ python -u "c:\Users\Row
Please enter your age: 5
5
```

Think how this use of functions and global variables can be added to the previous exercises. Each of the exercises are well suited to be split into functions. Here is an example using “Fruit Talk”…



```
import random

fruits = ["banana", "apple", "pineapple", "pear", "berries"]
```

This list is a global variable!

```
def main_program():
```

This function is our main program

```
    name = str(input("Hello, what's your name?: "))

    print("Hello " + name)

    likes_fruit = str.lower(input("Do you like fruit " + name + "?(yes/no): "))
    while(not(likes_fruit == "yes" or likes_fruit == "no")):
        likes_fruit = str.lower(
            input("Do you like fruit " + name + "?(yes/no): "))

    if(likes_fruit == "yes"):
        favourite_fruits = []
        likes_random = str()  # it's a string
        # while it isnt "stop" and the list still has choices
        while(not(likes_random == "stop") and len(fruits) > 0):
            random_fruit = randomFruit()  # get a random fruit
            likes_random = str.lower(
                input("Do you like " + random_fruit + "?(yes/no/stop): "))  # ask about it
            if likes_random == "yes":
                # add it to the list if they say yes
                favourite_fruits.append(random_fruit)

            # otherwise remove it to never ask again and loop back around to the top.
            fruits.remove(random_fruit)

        print("Ok we'll stop! Here's what you like: ")
        print(favourite_fruits)

    else:
        print("Oh, wow! I like some fruits but not all of them.")


def randomFruit():
```

This function when called acts as **`random.choice(fruits)`**

```
    return random.choice(fruits)


main_program()
```

Calling the program

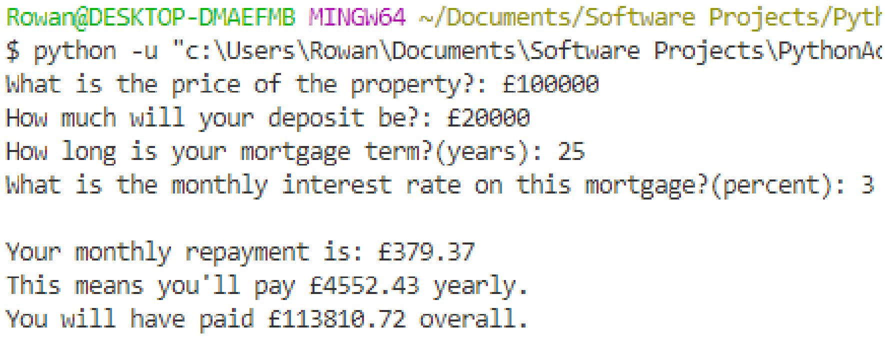
Find more on Functions and other resources in the endnotes.[xi]

Bonus Exercise

Mortgage Calculator

We'll be calculating a **fixed rate** mortgage **repayment + interest** calculator. A good solution will provide accurate figures as to monthly repayments. **This does not look at mortgage fees, legal fees or stamp duty tax.**

Task: Take the mortgage details from the user and compute their monthly, yearly, and overall repayment amounts.



```
Rowan@DESKTOP-DMAEFMB MINGW64 ~/Documents/Software Projects/Pytl
$ python -u "c:\Users\Rowan\Documents\Software Projects\PythonA
What is the price of the property?: £100000
How much will your deposit be?: £20000
How long is your mortgage term?(years): 25
What is the monthly interest rate on this mortgage?(percent): 3

Your monthly repayment is: £379.37
This means you'll pay £4552.43 yearly.
You will have paid £113810.72 overall.
```

Useful links (you'll need to recognise which pieces of code are relevant to our task):

- Code in Python and the underlying math.
- Graphical system tutorial including the Python code for the functions

Extension Ideas - There are no strict challenges for this bonus exercise, however if you wish you can consider the following…

Novice idea: Show repayments given a variable increase in interest. Similar to the Money Advice Service calculator.[xii]

Expert idea: Predict appreciation of the property based on the UK average for the past year. You can make further predictions using past data and attempt to look at the predicted value of the house after it has been repaid – allowing the user to evaluate the risk/reward of taking on these mortgage terms.

Notes

[i] How to uninstall in Windows (uninstall Python before new install) https://www.lifewire.com/windows-uninstaller-to-remove-unused-programs-3506955
[ii] Adding Python to PATH manually if all else fails https://datatofish.com/add-python-to-windows-path/
[iii] macOS tutorial and images https://www.javatpoint.com/how-to-install-python-on-mac
[iv] What is a text editor? https://en.wikipedia.org/wiki/Text_editor
[v] VSCode key bindings https://code.visualstudio.com/docs/getstarted/keybindings
[vi] Python documentation https://docs.python.org/3/
[vii] English conditional sentences https://en.wikipedia.org/wiki/English_conditional_sentences
[viii] Python operators https://www.w3schools.com/python/python_operators.asp
[ix] Vanguard pension calculator https://www.vanguardinvestor.co.uk/what-we-offer/personal-pension/pension-calculator
[x] Income tax calculator https://www.moneysavingexpert.com/tax-calculator/
[xi] Python Functions https://www.w3schools.com/python/python_functions.asp
[xii] Mortgage calculator https://www.moneyadviceservice.org.uk/en/tools/mortgage-calculator

GitHub Repository (Containing all code and examples) https://github.com/RowanAldean/Python-Basics-Short-Course