

Fast-track Python

Sample manual - first two chapters



TABLE OF CONTENTS (1 of 7)

1	GETTING STARTED	Page
1.1	Introducing Python	9
	<i>Installing Python</i>	9
1.2	Choosing an Editor	10
	<i>Choices of Editor</i>	10
1.3	Using IDLE	10
	<i>Running Single Commands (Interactive Mode)</i>	11
	<i>Colour-Coding / Case Sensitivity</i>	11
	<i>Creating, Saving and Running Programs</i>	12

2	BASIC CODING	Page
2.1	Comments	13
	<i>Single-line Comments</i>	13
	<i>Multi-line Comments</i>	13
	<i>Commenting Out Lines (and Uncommenting)</i>	13
2.2	Variables	13
	<i>Declaring Variables</i>	14
	<i>Variable Types</i>	15
	<i>Determining Type</i>	15
	<i>Assigning Values to Variables</i>	16
	<i>Deleting Variables</i>	16
2.3	Rules of Arithmetic	17
2.4	Basic Strings	18
	<i>New Lines and Tabs</i>	18
	<i>Quotation Marks</i>	18
	<i>Backslash Characters</i>	18
	<i>Concatenating Text</i>	19
	<i>Converting Numbers to Text</i>	19
	<i>Getting Inputs from Users</i>	19
2.5	Testing Conditions	20
	<i>Simple Conditions</i>	20
	<i>Multiple Conditions</i>	20
	<i>Testing for Equality</i>	21
	<i>Combining and Negating Conditions</i>	21

3	VISUAL STUDIO CODE	Page
3.1	Installing Visual Studio Code	22
3.2	The Visual Studio Code Window	23
	<i>The Activity Bar</i>	23
	<i>Changing your Theme</i>	23
3.3	Installing Extensions	24
	<i>Installing Extensions (using Python as an Example)</i>	24
3.4	Using Terminal Window	25
	<i>Viewing Terminal Windows</i>	25
	<i>Interactive Python Sessions</i>	25
3.5	Configuring VS Code Settings	26
	<i>Changing Settings using The Command Palette</i>	26
	<i>Changing Settings using the Menu</i>	26
	<i>Using Settings (JSON)</i>	27
	<i>Automatically Showing Default Settings</i>	27
	<i>Copying Default Settings to Customise Them</i>	28
	<i>Typical Default Settings</i>	28
3.6	Other Useful VS Code Tips	29
	<i>Commenting and Uncommenting Code</i>	29
	<i>Using Multiple Insertion Points</i>	29
	<i>Global Changes using Multiple Insertion Points</i>	30
	<i>Entering and Leaving Zen Mode</i>	30
	<i>Expanding and Collapsing Code</i>	31
	<i>Restoring Default Zoom</i>	31

4	WRITING AND RUNNING PROGRAMS	Page
4.1	Files and Folders	32
	<i>Opening a Folder</i>	32
	<i>Creating Python Files</i>	32
4.2	Running Programs	33
	<i>Clearing the Terminal Window</i>	33
	<i>Three Ways to Run a Program</i>	33
4.3	Basic Debugging	34
	<i>Setting and Unsetting Breakpoints</i>	34
	<i>Debugging</i>	34
4.4	Terminal Input (Revisited)	35
4.5	The Code Runner Extension	36
	<i>Installing the Extension</i>	36
	<i>Running Programs</i>	36
	<i>Changing the Run Key Combination</i>	37
	<i>Customising Code Runner</i>	38

TABLE OF CONTENTS (2 of 7)

5	VIRTUAL ENVIRONMENTS	Page
5.1	What Virtual Environments Are	39
5.2	Creating a Virtual Environment	40
	<i>Creating and Opening a Folder</i>	40
	<i>Creating a Virtual Environment</i>	41
	<i>Structure of a Virtual Environment</i>	41
	<i>Activating a Virtual Environment</i>	42
	<i>Selecting an Interpreter</i>	43

6	IMPORTING MODULES	Page
6.1	Importing Modules	44
	<i>Importing a Module</i>	44
	<i>Giving Modules Aliases</i>	45
	<i>Importing Specific Functions</i>	45
	<i>Importing Functions and Using Aliases</i>	45
6.2	Some Useful Built-In Modules	46
6.3	Using External Modules	47
	<i>Installing a Module</i>	47
	<i>Using Installed Modules</i>	48
	<i>Module Not Found Error</i>	48
	<i>Listing External Modules</i>	48
	<i>Viewing External Modules</i>	49

7	FORMATTING TEXT AND NUMBERS	Page
7.1	Basic Ways to Format Output	50
	<i>Using the F Prefix</i>	50
	<i>Using the Format Function</i>	51
	<i>Placeholder Order can Vary or be Omitted</i>	51
7.2	Formatting Numbers	52

8	RANGES AND LOOPS	Page
8.1	While Loops	53
	<i>Syntax of the While Command</i>	53
	<i>Example of a While Loop</i>	54
	<i>Using Else with While</i>	54
8.2	Break, Continue and Pass	55
8.3	For Loops	56
8.4	Ranges	57

9	DEBUGGING	Page
9.1	Overview	58
9.2	Preparing to Debug	59
	<i>Step 1 – Creating a Configuration File</i>	59
	<i>Step 2 - Setting a Breakpoint</i>	60
	<i>Step 3 – Turning off the JustMyCode Flag</i>	60
9.3	Debugging	61
9.4	Viewing and/or Changing Variable Values	62
	<i>The Variables Pane</i>	62
	<i>Watching Variables and Expressions</i>	62
	<i>The Debug Console</i>	63
9.5	Breakpoints	64
	<i>Conditional Breakpoints</i>	64
	<i>Disabling Breakpoints</i>	65
	<i>Deleting Breakpoints</i>	65
	<i>Removing or Disabling All Breakpoints</i>	65
9.6	Debugging Function Calls	66
	<i>Stepping Into, Over and Out of Functions</i>	66
	<i>The Call Stack</i>	67
	<i>Function Breakpoints</i>	67
9.7	Logging Breakpoints	68

TABLE OF CONTENTS (3 of 7)

10	SEQUENCES	Page
10.1	Introduction to Sequences	69
	<i>Main Types of Sequences in Python</i>	69
	<i>Reminder of Iterating Over Sequences</i>	69
10.2	Tuples versus Lists	70
	<i>Mutability (Lists versus Tuples)</i>	70
10.3	Slicing Sequences	71
	<i>Examples of Slicing for Lists</i>	71
	<i>Examples of Slicing for Ranges</i>	72
	<i>Examples of Slicing for Strings</i>	72
	<i>Missing Items when Slicing (Step Values)</i>	73
10.4	Joining and Splitting Sequences	74
	<i>Joining Sequences Together</i>	74
	<i>Concatenating Sequence Members</i>	74
	<i>Splitting Strings to Generate Sequences</i>	75
	<i>Splitting a String into Before and After Text</i>	75
10.5	Unpacking Sequences	76
10.6	Working with Sequences	77
	<i>Getting the Length of a Sequence</i>	77
	<i>Getting the Number of Items of a Specific Value</i>	77
	<i>Aggregating a Sequence's Items</i>	77
	<i>Getting the Index Number of an Item</i>	78
	<i>Returning Sequence Index Numbers and Values</i>	78
	<i>Mixing Data Types</i>	79
10.7	Examples of Sequences	80
	<i>Listing the Files in a Folder (ListDir)</i>	80
	<i>Listing the Files in a Folder using Glob</i>	80
	<i>Dividing Text into Lists of Words or Phrases</i>	81
	<i>A Tuple Listing Built-In Module Names</i>	81
	<i>Scraping Websites for Links</i>	82

11	MANIPULATING LISTS	Page
11.1	Adding and Removing Items	83
	<i>Inserting Items</i>	83
	<i>Appending to and Extending Lists</i>	84
	<i>Removing Items from Lists by Value</i>	84
	<i>Popping Items from a List by Position</i>	85
	<i>Clearing the Contents of Lists</i>	85
11.2	Changing the Order of Lists	86
	<i>Sorting Lists</i>	86
	<i>Reversing Lists</i>	86
11.3	Shallow and Deep Copying of Lists	87
	<i>Assigning is not Copying</i>	87
	<i>Shallow Copying</i>	87
	<i>Deep Copying</i>	88

12	COMPREHENSIONS AND GENERATORS	Page
12.1	Comprehensions	89
	<i>Basic Comprehensions</i>	89
	<i>Comprehensions with Conditions</i>	90
	<i>Multiple Loops within Comprehensions</i>	90
12.2	Generators	91
	<i>Disadvantages of Generators</i>	91

13	FILES AND FOLDERS	Page
13.1	Writing to Text Files	92
13.2	Using With to Close Files Automatically	93
13.3	Reading Files	94
	<i>Checking if Files and Folders Exist</i>	94
	<i>Reading Line by Line or Reading Characters</i>	94
	<i>Reading All the Lines in a File using Readlines</i>	95
	<i>Reading All the Lines in a File by Looping</i>	95
13.4	Looping Over Files	96
	<i>Looping Over Files in a Folder</i>	96
	<i>Processing Files in a Folder</i>	97
	<i>Looping Recursively</i>	97

14	ERROR-HANDLING	Page
14.1	Trapping for Errors	98
	<i>Error Types</i>	98
	<i>Trapping General Errors</i>	98
	<i>Trapping Specific Errors</i>	99
	<i>The Full Range of Commands</i>	99
14.2	Raising Exceptions	100

TABLE OF CONTENTS (4 of 7)

15	NUMBERS, STRINGS AND DATES	Page
15.1	Overview	101
15.2	Working with Numbers	102
	<i>Mathematical Operators</i>	102
	<i>Built-in Numerical Functions</i>	102
	<i>Math Functions</i>	103
15.3	Working with Boolean Values	104
	<i>Boolean Operators</i>	104
	<i>All and Any</i>	104
15.4	Working with Dates (and Times)	105
	<i>Getting Dates (and Times)</i>	105
	<i>Formatting Dates</i>	106
	<i>Formatting Times</i>	106
	<i>Displaying Calendar Months</i>	107
	<i>Displaying Day and Month Names</i>	107
15.5	Working with Strings	108
	<i>Escape Characters</i>	108
	<i>Avoiding Escape Characters</i>	108
	<i>Joining and Splitting Text</i>	109
	<i>Repeating Text</i>	109
	<i>Extracting Text (Slicing)</i>	109
	<i>Counting and Length</i>	110
	<i>Changing Case</i>	110
	<i>Padding</i>	110
	<i>Removing and Replacing Text</i>	111
	<i>Translating Text</i>	111
	<i>Finding Text</i>	112
	<i>Checking Text Content</i>	113

16	SETS	Page
16.1	Some Set Concepts	114
16.2	Working with Sets	115
	<i>Creating Sets</i>	115
	<i>Set Operations</i>	115
16.3	Converting between Sets and Lists	116
	<i>Converting Sets to Lists</i>	116
	<i>Converting Lists to Sets</i>	117
16.4	Examples of the Use of Sets	118
	<i>Counting Unique Letters or Words</i>	118
	<i>Finding the Differences between Lists</i>	119

17	DICTIONARIES	Page
17.1	Creating Dictionaries	120
	<i>What is a Dictionary?</i>	120
	<i>Creating Dictionaries</i>	120
17.2	Using Dictionaries	121
	<i>Looking Up Items</i>	121
	<i>Looping Over Dictionary Items</i>	121
	<i>Adding, Editing and Deleting Items</i>	122
	<i>Sorting Dictionaries</i>	122

18	WRITING FUNCTIONS	Page
18.1	The Need for Functions	123
	<i>Advantages of Using Functions</i>	123
18.2	Writing a Function	124
	<i>Step 1 – Identifying the Input Arguments</i>	124
	<i>Step 2 – Specifying the Output Data Type</i>	124
	<i>Step 3 – Reviewing the Syntax Required</i>	125
	<i>Step 4 – Writing your Functions</i>	125
18.3	Learning Points	126
	<i>Variable Names are Isolated</i>	126
	<i>Arguments can have Different Names</i>	126
	<i>Functions can be Declared in any Order</i>	127
	<i>Your Function could Crash in Many Ways</i>	127
	<i>Data Types are for Guidance Only</i>	128
18.4	Ways to Pass Arguments	129
	<i>Arguments by Name or Position</i>	129
	<i>Forcing Positional or Named Arguments</i>	130
	<i>Optional Arguments</i>	130
18.5	Arbitrary and Keyword Arguments	131
	<i>Passing an Unknown Number of Argument Values</i>	131
	<i>Passing an Arbitrary Set of Arguments</i>	132
18.6	Using Modules for Functions	133
18.7	Modular Programming	134
18.8	Docstrings	135

TABLE OF CONTENTS (5 of 7)

19	SCRAPING WEBSITES	Page
19.1	Case Study	136
19.2	Preparation for Web Scraping	137
	<i>Understand the Underlying HTML</i>	137
	<i>Understanding the Limitations</i>	137
19.3	Understanding HTML	138
	<i>The Document Object Model (DOM)</i>	138
	<i>HTML Tags</i>	138
	<i>Tag Attributes</i>	139
	<i>Classes and Ids</i>	139
19.4	Getting Started	140
	<i>Choosing a Web Scraping Tool</i>	140
	<i>Limitations of the Requests Module</i>	140
	<i>Installing BeautifulSoup</i>	141
	<i>Getting Help</i>	141
19.5	Getting HTML	142
	<i>Browsing a Web Site</i>	142
	<i>Possible HTML Parsers</i>	142
	<i>Using HTML from File</i>	143
19.6	Ways to Navigate	144
	<i>Chaining Elements</i>	144
	<i>Getting a List of Relatives of an Element</i>	144
	<i>Example: Contents versus Descendants</i>	145
	<i>Example: Showing All Text</i>	145
19.7	Getting Output	146
19.8	Finding Elements	147
	<i>Finding Elements by Tag</i>	147
	<i>Finding Elements by Attribute</i>	148
	<i>Finding Attributes by Class</i>	148
	<i>Non-recursive Finds</i>	148
19.9	Searching Using CSS Selectors	149
	<i>Example: Finding Elements of Given Class</i>	149
	<i>Finding Elements within a Given Id</i>	149

20	WORKING WITH EXCEL	Page
20.1	Getting Started with Openpyxl	150
	<i>Installing Openpyxl</i>	150
	<i>Getting Help with OpenPyXl</i>	150
20.2	Working with Workbooks	151
	<i>Creating and Saving Workbooks</i>	151
	<i>Opening and Closing Workbooks</i>	151
20.3	Working with Worksheets	152
	<i>Inserting Worksheets</i>	152
	<i>Our Example Workbook</i>	152
	<i>Getting a List of Worksheet Names</i>	153
	<i>Getting a Worksheet Itself</i>	153
	<i>Getting and Setting the Active Worksheet</i>	153
	<i>A Worked Example</i>	154
	<i>Looping over Worksheets</i>	154
20.4	Working with Cells	155
	<i>Referring to Single Cells</i>	155
	<i>Useful Cell Properties</i>	155
20.5	Looping over Cells	156
	<i>Looping over Row and/or Column Numbers</i>	156
	<i>Offsetting Cells</i>	156

21	OTHER FILE TYPES	Page
21.1	Overview	157
21.2	Linking to SQL Server (or Other Databases)	158
	<i>Importing Pyodbc</i>	158
	<i>Creating a Connection</i>	158
	<i>Creating a Cursor</i>	158
	<i>Choosing a Fetch Method</i>	159
	<i>A Full Worked Example</i>	160
	<i>Inserting, Updating and Deleting Data</i>	161
	<i>Running Stored Procedures with Parameters</i>	161
21.3	CSV Files	162
	<i>Writing CSV Files</i>	162
	<i>Reading CSV Files</i>	162
21.4	JSON Files	163
	<i>Serialisation and Deserialisation</i>	163
	<i>Conversions</i>	163
	<i>Loading and Dumping</i>	164
	<i>Writing to JSON Files</i>	164
	<i>Reading from JSON Files</i>	165
	<i>Reading an External JSON File</i>	165

TABLE OF CONTENTS (6 of 7)

22	NUMPY	Page
22.1	Overview	166
	Our Example	166
	Advantages of NumPy	166
	Some NumPy Terms	167
22.2	Creating Arrays	168
	Data Types	168
	Standard Arrays	168
	Filling Arrays Automatically	169
	Filling Arrays with Pre-set Values	169
	Filling Arrays with Random Numbers	170
	Filling Arrays from Sequences	170
22.3	Working with Parts of an Array	171
	Slicing Arrays	171
	Another Slicing Example	172
22.4	Array Maths	173
	Aggregating	173
	Array Multiplication and Broadcasting	174
	Matrix Multiplication	175
	Scalar (Elementwise) Operations	175
22.5	Operations on Arrays	176
	Transposing an Array	176
	Flattening or Ravelling an Array	176
	Changing Array Elements	177
	Reshaping and Resizing Arrays	177
	Joining Arrays (Stacking and Concatenating)	178
	Other Array Operations	178

23	PANDAS	Page
23.1	Overview	179
	Why Use Pandas not Excel?	179
23.2	Creating Dataframes and Series	180
	Creating Dataframes	180
	Creating Series	180
23.3	Reading and Writing Dataframes	181
	Reading from Excel or CSV Files	181
	Reading in a SQL Server Table	181
	Writing to CSV	182
	Writing to Excel	182
	Writing to a SQL Server Table	183
23.4	Showing Dataframe Information	184
23.5	Showing Parts of a Dataframe	185
	Top and Bottom Rows (Heads and Tails)	185
	Picking Out Particular Series or Columns	185
	Filtering Dataframes	186
	Slicing Rows using ILOC	186
	Returning Indexed Rows Only using LOC	186
23.6	Aggregating Data	187
	Single Statistics	187
	Statistics for Multiple Columns	187
	Showing Multiple Statistics	188
	Showing Grouped Statistics	188
23.7	Calculations with Columns	189
	Getting at the Text in a Column	189
	Mathematical Calculations on Columns	189
	Working with Dates	190
23.8	Other Dataframe Operations	191
	Renaming Columns	191
	Sorting Dataframes	191
	Joining Dataframes Together	192
	Removing Duplicates	192
23.9	Plotting Dataframes	193

24	POWER BI AND PYTHON	Page
24.1	Preparing for Python	194
24.2	Getting Data from Python	195
	Creating your Dataframe	195
	Starting to Create a Python Data Source	195
	Finishing the Process	196
24.3	Power BI Python Visuals	197
	Starting your Visual	197
	Creating the Visual	198
	Seeing the Full Code	198

TABLE OF CONTENTS (7 of 7)

25	PYTHON CODING USING AI TOOLS	Page
25.1	Choosing an AI Tool	199
25.2	Generating Code	200
	<i>Our Example – Scraping a Website</i>	200
	<i>A Critique of the Code Generated</i>	201
	<i>Problems with the Code</i>	201
	<i>Simplifying the Code</i>	202
25.3	Refactoring / Changing Code	203
	<i>Global Variable Changes</i>	203
	<i>Stylistic Changes</i>	203
25.4	Optimising Code	204
	<i>Writing Code more Efficiently</i>	204
	<i>Changing the Algorithm</i>	204
25.5	Debugging	205
	<i>Our Example – Reading a Shopping List</i>	205
	<i>What to Ask</i>	205
	<i>Listing the Bugs</i>	206
25.6	Researching Modules	207

CHAPTER 1 - GETTING STARTED

1.1 Introducing Python

Python is the only programming language named after a BBC comedy series. It was originally created by a Dutch programmer called Guido van Rossum.



The author has programmed extensively in VB, C# and SQL, and is an enthusiastic convert to Python. It will let you develop powerful programs quickly, although its management of paths and packages will have you tearing your hair out!

Installing Python

You can download Python from <https://www.python.org/downloads/> :

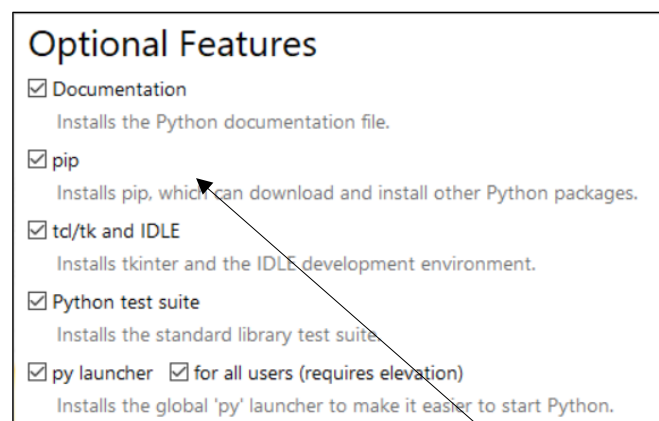
Click on this button to install the latest version of Python at the time of writing.



Here are our recommended settings:



a) Managing paths in Python is a pain! Do yourself a favour and tick this box to help Windows programs run Python easily.



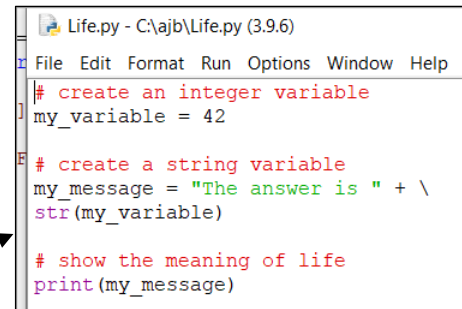
b) You can choose which bits of Python you want to install, although it's probably best just to go with the defaults.

You'll certainly want to install **pip** (which will help you import modules to accomplish different tasks), **td/tk** (which will help you create GUI applications) and **IDLE** (a Python editor – see next page).

1.2 Choosing an Editor

Python comes with its own built-in editor called IDLE (named after Eric), but it's a bit primitive:

A program written in IDLE. Although it's a great package for getting started, it doesn't have true Intellisense (which in this owl's view rules it out as a serious development environment).



```
Life.py - C:\ajb\Life.py (3.9.6)
File Edit Format Run Options Window Help
# create an integer variable
my_variable = 42
# create a string variable
my_message = "The answer is " + \
str(my_variable)
# show the meaning of life
print(my_message)
```

Choices of Editor

Here are some possible editors that you could choose:

Editor	Notes
<i>Visual Studio Code</i>	A generic code editor maintained by Microsoft but available free of charge (don't confuse it with Visual Studio, which is a completely different program – see below).
<i>PyCharm</i>	An editor devoted to writing Python code. Reviews online suggest that it can be very slow to work with, and some users will need to upgrade to the paid Premium edition.
<i>Visual Studio</i>	If you already spend time working in Visual Studio, you may find it easiest to use this as your development environment (although it's a bit of a big beast!).
<i>Jupyter Notebooks</i>	If you work in machine learning or AI you may well choose this powerful coding environment.



There are many other Python editors out there with names like Atom and Sublime, as well as tools which will manage your Python code such as Anaconda. This courseware uses IDLE to get started, then switches to Visual Studio Code.

1.3 Using IDLE

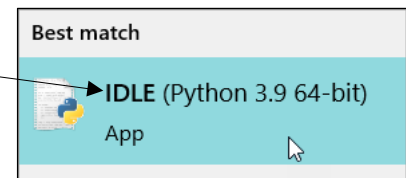
IDLE was (allegedly) named after Eric Idle, one of the Monty Python team. There's also a Python editor called *Eric*, but none called *Cleese* or *Palin* that Wise Owl know of.

The acronym does work well, it has to be said.



When you install Python, you should automatically get IDLE at the same time:

The latest version of the IDLE Python editor (at the time of writing).



Running Single Commands (Interactive Mode)

You can run any single command by typing it in at the command prompt and pressing :

```

IDLE Shell 3.9.6
File Edit Shell Debug Options Win
Python 3.9.6 (tags/v3.9.6:db3ff76,
D64) on win32
Type "help", "copyright", "
>>> |
  
```

a) The `>>>` text is called the command prompt - it's waiting for you to type in a valid Python command.

b) The print command just displays the information in parentheses in IDLE.

c) The output of your command is the message that you chose to print.

```

*IDLE Shell 3.9.6*
File Edit Shell Debug Options Window Help
Python 3.9.6 (tags/v3.9.6:db3ff76,
D64) on win32
Type "help", "copyright", "credits"
>>> print("Tu-whit, tu-whoo")
  
```

```

IDLE Shell 3.9.6
File Edit Shell Debug Options Window
Python 3.9.6 (tags/v3.9.6:db3ff76,
D64) on win32
Type "help", "copyright", "credits"
>>> print("Tu-whit, tu-whoo")
Tu-whit, tu-whoo
>>>
  
```

Colour-Coding / Case Sensitivity

Note that Python is a case-sensitive language!

Here IDLE hasn't colour-coded the word **Print**, because it doesn't recognise it as a valid Python command ...

... and displays an error when you press to try to run the command.

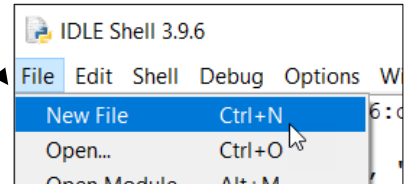
```

>>> Print("Tu-whit, tu-whoo")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    Print("Tu-whit, tu-whoo")
NameError: name 'Print' is not defined
>>> |
  
```

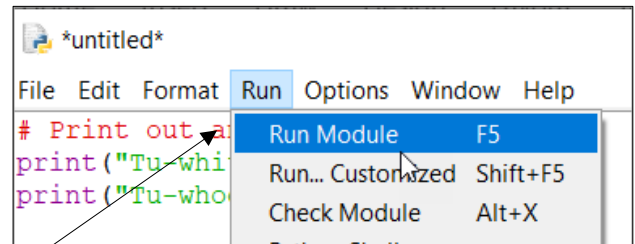
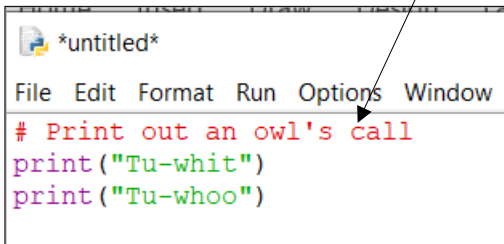
Creating, Saving and Running Programs

If you want to execute a sequence of commands, you don't have to run each one individually; instead you can save them in a *file*:

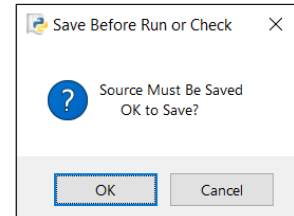
- a) Choose to create a new file from the **File** menu (as this shows, you can instead press **Ctrl + N**).



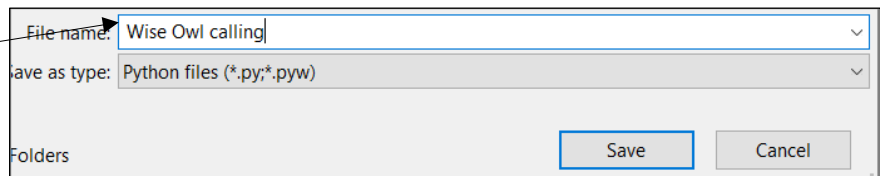
- b) Type in a sequence of valid Python statements (the **#** symbol denotes a comment, which will be ignored – more on comments shortly).



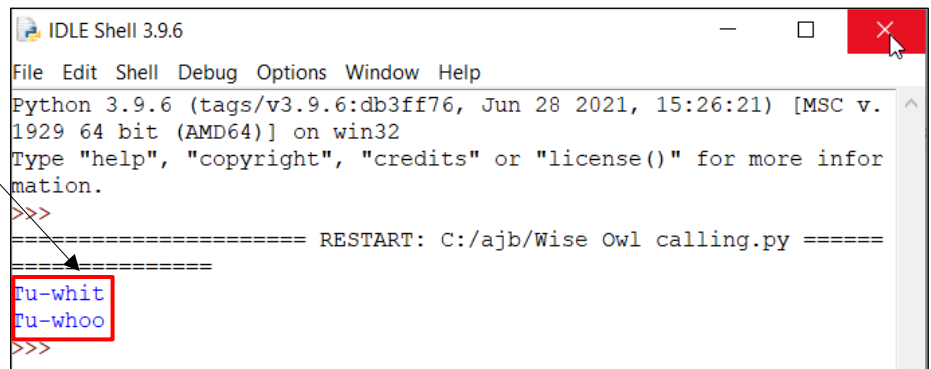
- c) Choose this option to run your code (although it's quicker to press **F5** instead as a short-cut). You'll be prompted to save your code in a file – until you do this you won't be able to run it.



- d) Your file will be saved with an extension of **.py**.



- e) You can now see the output from your program. When you've finished, click on the cross at the top right to close down the IDLE shell and return to your program.



You can obviously open existing files that you've previously created to run their code instead.

CHAPTER 2 - BASIC CODING

2.1 Comments

Good programmers add *comments* to their code, to explain what it is meant to do!

Single-line Comments

Most Python comments begin with a single `#` character:

Each separate line has to begin with its own `#` character. The red lines in this program will be completely ignored by the Python interpreter.

```
# A simple program to print
# out a wise owl's call
# using two separate lines

# Here's the first line ...
print("Tu-whit")

# ... and here's the second
print("Tu-whoo")
```

Multi-line Comments

You can use three double-quotation marks in a row to mark out multiple comment lines:

These lines will be treated as comments (confusingly, IDLE chooses to show them in green, not red).

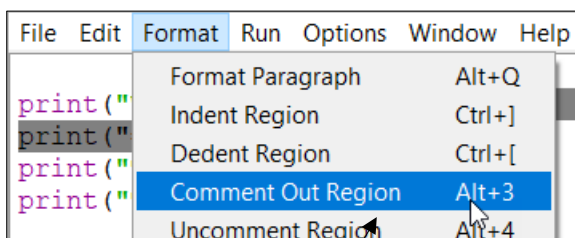
```
"""
A simple program to print
out a wise owl's call
using two separate lines
"""

# Here's the first line ...
print("Tu-whit")

# ... and here's the second
print("Tu-whoo")
```

Commenting Out Lines (and Uncommenting)

If you want to avoid running certain lines without removing them, you can *comment them out*:



a) Select part or all of the lines you want to comment out and press `Alt` + `3` or choose this menu option.

```
##print("What noise do owls make?")
##print("=====")
print("Tu-whit")
print("Tu-whoo")
```

b) For some reason IDLE puts two hashes in front of each commented out line.

You can select commented out lines and press `Alt` + `4` to reinstate them (or choose the **Uncomment Region** menu option shown above).

2.2 Variables

A *variable* is a space inside your computer which holds a single bit of information (be it a number, date, string of text or other value).

Two examples of Python variables.

```
# a variable to hold a company name
company_name = "Wise Owl Training"

# a variable to hold a number
answer = 42
```



The Python naming convention is to avoid camel case but instead use underscores to divide the parts of a variable name. Thus you might call the second variable above *meaning_of_life*, but wouldn't call it *MeaningOfLife*.

Declaring Variables

Here's how you declare an integer variable in 4 commonly used programming languages:

Language	Variable declaration
C#	<pre>// integer variable int answer = 42;</pre>
SQL	<pre>-- declare an integer variable DECLARE @answer int = 42</pre>
JavaScript	<pre>// declare a variable var answer = 42;</pre>
Visual Basic	<pre>'create integer variable Dim answer As Integer = 42</pre>

In Python, by contrast, you don't declare a variable before using it (the act of assigning a value to a variable automatically declares it at the same time).

```
# an integer variable to hold a number
answer = 42

# now print out this value (note misprint)
print(answe)
```

Here we've created a variable called **answer**, but are trying to print out the value of an (uncreated) variable called **answe**.

```
Traceback (most recent call last):
  File "P:/Manuals/new/Python/Files/Var
    print(type(answe))
NameError: name 'answe' is not defined
```

The Python interpreter points out the error of your ways when you try to run your program.



For experienced programmers in other languages this will be one of the weirdest things to get used to about Python, but it's a good idea (to the extent that you will now probably resent having to formally declare variables in other languages).

Variable Types

Here are some of the common data types in Python:

Data type	What it contains
str	Any string of text
int	Any whole number
float	Any decimal number
bool	Something which can be either true or false



*It is typical of Python that the old **long** type used to denote very large integers is no longer needed, and **int** covers everything from 0 to infinity!*

Determining Type

Python determines the type of a variable from the value you assign to it. You can see this by using the **type** function to investigate a variable's data type:

```
# set variable to hold a number
some_variable = 42
print(type(some_variable))

# change it to hold text
some_variable = "Wise Owl Training"
print(type(some_variable))

# now store a floating-point number
some_variable = 3.14
print(type(some_variable))

# now store a true/false value
some_variable = True
print(type(some_variable))
```

This program will assign different values to the same variable. After assigning each value, we print out the type of the variable.

Here's what this program would output:

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'bool'>
>>> |
```


Assigning Values to Variables

You can assign values to individual variables as we've already seen by using this convention:

```
variable_name = value_for_variable
```

If you're assigning two or more variables to the same value, you can do this in a single line:

```
# three variables to hold the same thing
bird_name = company_mascot = pooh_friend = "Owl"

# change the text of one of them
company_mascot = 'Wise Owl'

# prove this by printing out values
print(bird_name)
print(company_mascot)
print(pooh_friend)
```

This code would create 3 variables, but they would all contain the same value if we hadn't subsequently changed the value of one of them.

```
Owl
Wise Owl
Owl
>>>
```

You can do multiple assignments on the same line (although this owl thinks it makes your code harder to read):

```
# create 3 variables in one line
company_name, answer, months_in_year = "Wise Owl", 42, 12

print(company_name)
print(answer)
print(months_in_year)
```

This code would create (then show the values of) one string and two integer variables:

```
Wise Owl
42
12
>>>
```

Deleting Variables

Python will delete any variables that you've created when the program containing them finishes, but sometimes you may want to pre-empt this. You can do this using the **del** command:

```
# a string variable to hold a company name
company_name = "Wise Owl Training"

# an integer variable to hold a number
answer = 42

# delete both variables
del answer, company_name

# print out their values (this will crash)
print(company_name)
print(answer)
```

Once you've deleted a variable, not surprisingly you can no longer refer to it. Running this program would give the following error:

```
NameError: name 'company_name' is not defined
>>>
```

2.3 Rules of Arithmetic

Python follows the same order of arithmetic operation as most other computer packages (taught in schools as *BODMAS*, standing for Brackets Of Division Multiplication Addition Subtraction).

```
# create 3 variables
first = 3
second = 5
third = 2

# perform two calculations
answer_using_bodmas = first + second * third
answer_using_brackets = (first + second) * third

# show answers
print(answer_using_bodmas)
print(answer_using_brackets)
```

You can use parentheses to change the default order of operation. This code would give the following output:

```
13
16
>>>
```

For the second calculation, Python sums the first two numbers before multiplying the result by the third one.

In addition to the standard operators of `+`, `-`, `*` and `/` you can also use these:

Operation	Operator	
Raising to the power	<code>**</code>	<p>The following code excerpt would return 2^{10}, or 1024:</p> <pre># base number and power base_number = 2 power = 10 # get answer answer = base_number ** power # show result print(answer)</pre>
Taking the remainder or modulus of a number	<code>%</code>	<p>The following code would return 1 (the remainder when you divide 22 by 7):</p> <pre># base numbers start_number = 22 divisor = 7 # get answer answer = start_number % 7 # show result print(answer)</pre>

2.4 Basic Strings

A later chapter will give much more details on the tricks and functions you can use when working with strings of text; this page just shows a few basic ones.

New Lines and Tabs

You can use the escape characters `\n` and `\t` to include new lines and tabs in your output:

```
# include tabs and a line break in name
signature = "Company\t\tWise Owl\nBusiness\tTraining"
print(signature)
```

This program would produce the following output:

```
Company           Wise Owl
Business          Training
>>> |
```



*Tabs are an unreliable way to align output, as the example above shows (the first line includes two tabs, but the second only one). A better way to align text is to use the **ljust**, **rjust** and/or **center** functions (covered in a later chapter)*

Quotation Marks

To create a string which includes quotation marks, either use an escape character or switch from double to single quotation marks (the second way seems easier):

```
# two ways to include quotes
owl_name_1 = "My name is 'Owl'"
owl_name_2 = "My name is \"Owl\""

# show the results
print(owl_name_1)
print(owl_name_2)
```

Both of these ways would embed quotation marks in the relevant strings of text, to give this output:

```
My name is 'Owl'
My name is "Owl"
>>> |
```

Backslash Characters

Since the escape character is a `\`, how can you include this in a string of text? The answer is to repeat it:

```
# show the file path
print("The file path is C:\\wiseowl\\python\\")
```

This program would give the following output:

```
The file path is C:\wiseowl\python\
>>>
```

Concatenating Text

Use the `+` symbol to join bits of text together:

This program would join the two variables together with a space between them, to give this:

```
Wise Owl
>>> |
```

```
# two string variables
start = "Wise"
end = "Owl"

# show company name
print(start + ' ' + end)
```

Converting Numbers to Text

You can not join a string with a number; instead, you must first convert the number to a string using the **str** function.

```
# set name and age of someone ...
your_name = "Bob"
your_age = 42

# ... and print this out (will crash)
print (your_name + ", you are " + \
your_age + " years old")
```

This program will crash because it is trying to join a string of text (**your_name**) with an integer (**your_age**):

```
print (your_name + ", you are " + \
TypeError: can only concatenate str (not "int") to str
>>>
```

Note the use of a backslash at the end of this line to act as a continuation character, allowing a single programming command to span multiple lines.

Here's a working version of the code above:

You must use the **str** function to convert numbers to text before joining them with other bits of text. This would give:

```
Bob, you are 42 years old
>>> |
```

```
# set name and age of someone ...
your_name = "Bob"
your_age = 42

# ... and print this out
print (your_name + ", you are " + \
str(your_age) + " years old")
```

Getting Inputs from Users

You can pause a program to ask a user to input values using the **input** function - for example:

```
# get someone's name
your_name = input("Enter your name ==> ")
your_age = input("Enter your age ==> ")

print (your_name + ", you are " + \
your_age + " years old")
```

Running this program (and inputting the values **Bob** and **42** at the prompts) would give this output:

```
Enter your name ==> Bob
Enter your age ==> 42
Bob, you are 42 years old
>>> |
```



Note that the **input** function always gives a string of text, so there's no need to convert the 42 above to a string before concatenating it with the user's name.

2.5 Testing Conditions

Simple Conditions

You can use the **if** statement to test conditions in Python, but you must follow it with a colon `:` and indentation:

In most languages you would indent your code at this point by pressing `Tab` to make it more readable. In Python this space is a vital part of your code, and without it you'll get a run-time error.

```
# get the user's age
age = input("Type your age ==>")

# warn user if too young (must convert
# string returned from input to int first)
if int(age) <= 18:
    print("Sorry you are too young")
```

Multiple Conditions

If you want to test whether a condition is true or false, use **else**:

```
# get the user's age
age = input("Type your age ==>")

# convert this to an integer
# (note that this will crash if it isn't)
age_as_int = int(age)

# test age to see whether under 18 or not
if age_as_int < 18:
    print("Sorry you are too young")
else:
    print("Welcome!")
```

This program will display a different message for someone under 18 than for an adult. Here's a typical output from running the program:

```
Type your age ==>21
Welcome!
>>>
```

For multiple conditions, use as many **elif** statements as you need:

Here the program tests (in this order):

- Whether the age is less than 18
- Whether the age is less than 40 (knowing that it can't be under 18, otherwise it would have passed the first test)
- Whether the age is more than 60

The program only prints out a welcome message for people who don't meet any of these conditions. A typical output might be:

```
Type your age ==>65
Sorry you are too old
>>> |
```

```
# get the user's age
age = input("Type your age ==>")

# convert this to an integer
# (note that this will crash if it isn't)
age_as_int = int(age)

# test age to see what to do
if age_as_int < 18:
    print("Sorry you are too young")
elif age_as_int < 40:
    print("Sorry you are too middle-aged")
elif age_as_int > 60:
    print("Sorry you are too old")
else:
    print("Welcome!")
```



From version 3.10 Python will have a powerful **match** statement giving the equivalent of the C#/JavaScript **switch** statement, the VB **SELECT CASE** statement or the SQL **CASE WHEN** statement, reducing the need for multiple **elif** statement blocks like the one shown above.

Testing for Equality

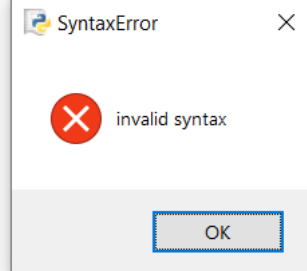
When you are testing if two values are equal in Python, you must use two `=` signs in a row.

This can be a very disconcerting error message to see – what could you possibly have done wrong? The answer is that because you're testing a condition you need to put this:

```
# are they the same?
if first_age == second_age:
    print("Same ages")
else:
    print("Different ages")
```

```
# get two people's ages
first_age = 42
second_age = 44

# are they the same?
if first_age = second_age:
    print("Same ages")
else:
    print("Different ages")
```



If you're not used to it, this Python feature (and the fact that everything is case-sensitive) will probably account for about 90% of the bugs that you create!

Combining and Negating Conditions

You can use the **and**, **or** and **not** keywords to test different combinations of conditions:

This program would output this::

```
Welcome (level 0)
Welcome (level 1)
Welcome (level 2)
>>>
```

This is because:

- This person is NOT coughing
- The person is either vaccinated OR masked (in fact, they're both)
- The person is masked AND vaccinated

```
# a (hopefully) nostalgic look back ...
if_vaccinated = True
if_masked = True
if_coughing = False

# a few places just require you to look well
if not if_coughing:
    print("Welcome (level 0)")

# can shop if either vaccinated or masked
if if_masked or if_vaccinated:
    print("Welcome (level 1)")

# some places require both
if if_masked and if_vaccinated:
    print("Welcome (level 2)")
```

Python also treats the following two statements as identical:

The second test also checks that 18 is less than or equal to the value of the variable **age** and that the value of the variable **age** is less than 65.

```
if age >= 18 and age < 65:
    if 18 <= age < 65:
```

Blank lined area for writing.



Blank lined area for writing.



Blank lined area for writing.







Blank lined paper for writing.



Blank lined paper for writing.



Blank lined paper for writing.






































Blank lined paper for writing.



WiseOwl
Training



What we do!

		Basic training	Advanced training	Systems / consultancy
Office	Microsoft Excel			
	VBA macros			
	Office Scripts			
	Microsoft Access			
Power BI, etc	Power BI and DAX			
	Power Apps			
	Power Automate (both)			
SQL Server	SQL			
	Reporting Services			
	Report Builder			
	Integration Services			
	Analysis Services			
Coding	Visual C#			
	VB programming			
	MySQL			
	Python			



WiseOwl
Training

