Fast-track Python

Sample manual - first two chapters

Wise Owl Training

TABLE OF CONTENTS (1 of 7)

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

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

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

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

© Copyright 2024

TABLE OF CONTENTS (2 of 7)

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

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

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

8	RANGES AND LOOPS	Page
8.1	While Loops	53
	Syntax of the While Command	53
	Example of a While Loop	54
	Using Else with While	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
	Step 1 – Creating a Configuration File	59
	Step 2 - Setting a Breakpoint	60
	Step 3 – Turning off the JustMyCode Flag	60
9.3	Debugging	61
9.4	Viewing and/or Changing Variable Values	62
	The Variables Pane	62
	Watching Variables and Expressions	62
	The Debug Console	63
9.5	Breakpoints	64
	Conditional Breakpoints	64
	Disabling Breakpoints	65
	Deleting Breakpoints	65
	Removing or Disabling All Breakpoints	65
9.6	Debugging Function Calls	66
	Stepping Into, Over and Out of Functions	66
	The Call Stack	67
	Function Breakpoints	67
9.7	Logging Breakpoints	68

© Copyright 2024

TABLE OF CONTENTS (3 of 7)

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

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

12	COMPREHENSIONS AND GENERATORS	Page
12.1	Comprehensions	89
	Basic Comprehensions	89
	Comprehensions with Conditions	90
	Multiple Loops within Comprehensions	90
12.2	Generators	91
	Disadvantages of Generators	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
	Checking if Files and Folders Exist	94
	Reading Line by Line or Reading Characters	94
	Reading All the Lines in a File using Readlines	95
	Reading All the Lines in a File by Looping	95
13.4	Looping Over Files	96
	Looping Over Files in a Folder	96
	Processing Files in a Folder	97
	Looping Recursively	97

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

© Copyright 2024

TABLE OF CONTENTS (4 of 7)

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

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

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

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

© Copyright 2024

TABLE OF CONTENTS (5 of 7)

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

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

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

© Copyright 2024

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

© Copyright 2024

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
	Our Example – Scraping a Website	200
	A Critique of the Code Generated	201
	Problems with the Code	201
	Simplifying the Code	202
25.3	Refactoring / Changing Code	203
	Global Variable Changes	203
	Stylistic Changes	203
25.4	Optimising Code	204
	Writing Code more Efficiently	204
	Changing the Algorithm	204
25.5	Debugging	205
	Our Example – Reading a Shopping List	205
	What to Ask	205
	Listing the Bugs	206
25.6	Researching Modules	207

© Copyright 2024

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!

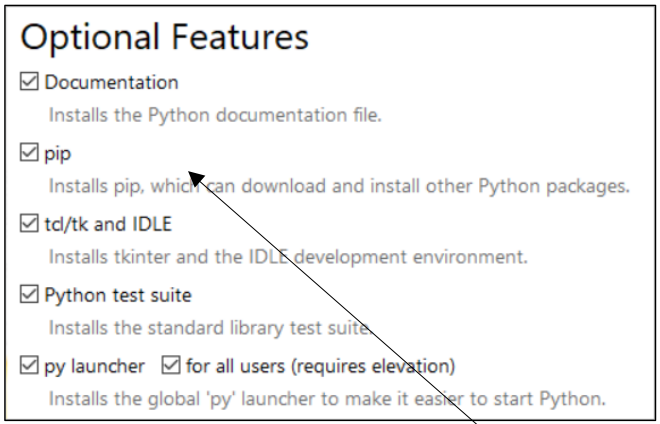
Installing Python

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.

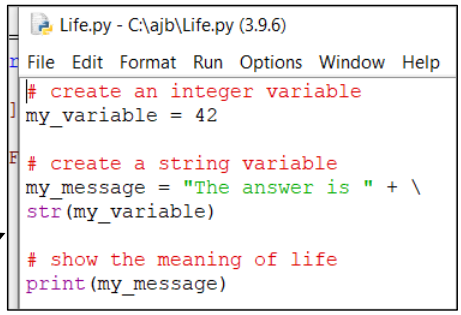


b) You can choose which bits of Python you want to install, although it's probably best just to go with the defaults.

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).

Choices of Editor

Here are some possible editors that you could choose:

Editor	Notes
Visual Studio Code	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).
PyCharm	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.
Visual Studio	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!).
Jupyter Notebooks	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.

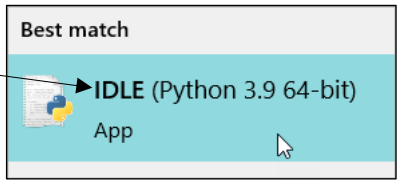
1.3 Using IDLE

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.

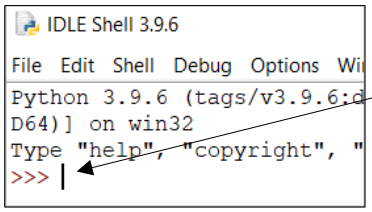
When you install Python, you should automatically get IDLE at the same time:

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



Running Single Commands (Interactive Mode)

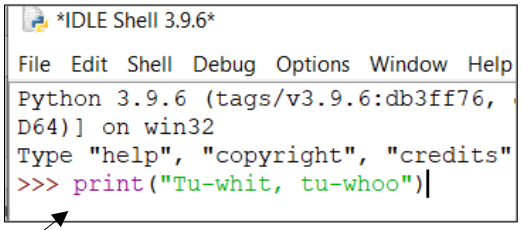
You can run any single command by typing it in at the command prompt and pressing ↵ :

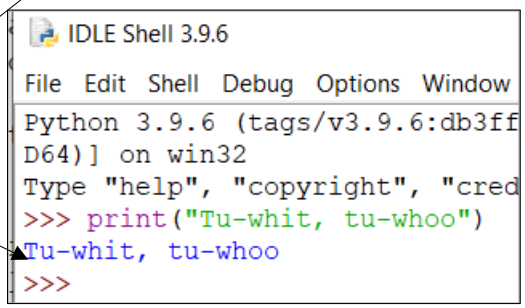


a) The >>> text is called the command prompt - it's waiting for you to type in a valid Python command.

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.



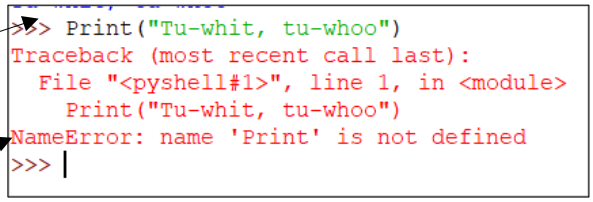


Colour-Coding / Case Sensitivity

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.

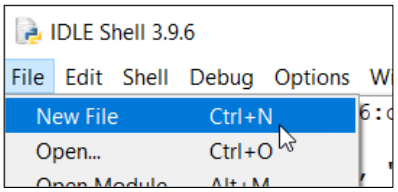


© Copyright 2024

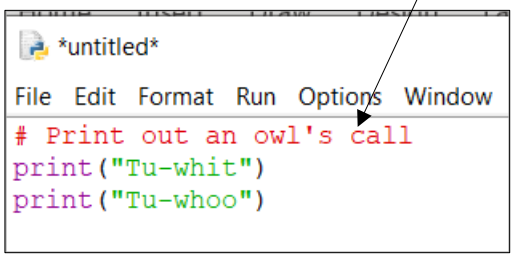
Creating, Saving and Running Programs

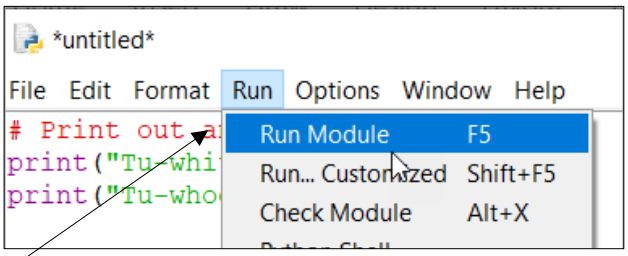
If you want to execute a sequence of commands, you don't have to run each one individually; instead you can save them in a *file*:

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

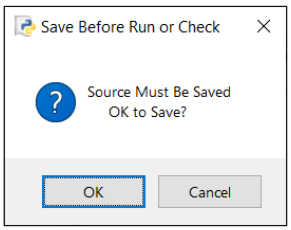


b) Type in a sequence of valid Python statements (the # symbol denotes a comment, which will be ignored – more on comments shortly).

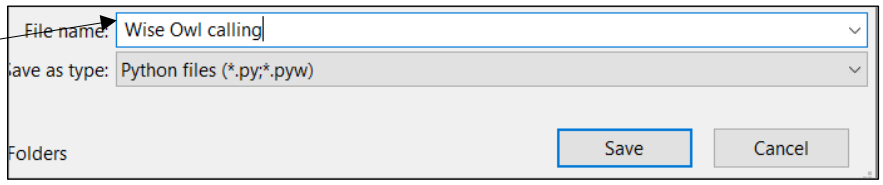




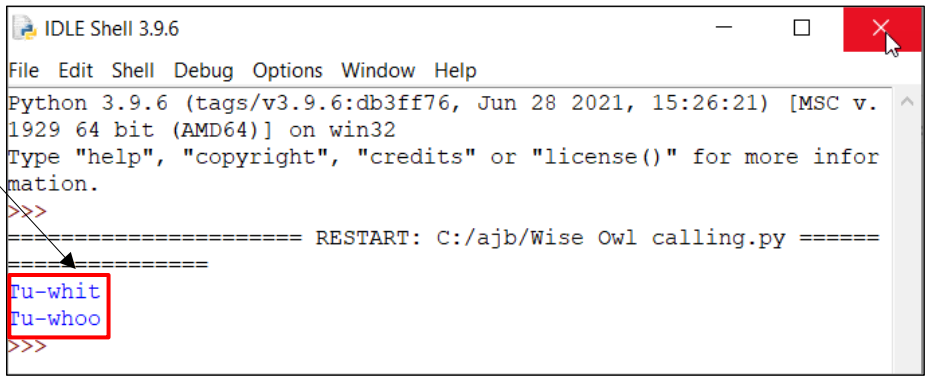
c) Choose this option to run your code (although it's quicker to press F5 instead as a short-cut). You'll be prompted to save your code in a file – until you do this you won't be able to run it.



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.

© Copyright 2024

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:

```
# 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")
```

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

Multi-line Comments

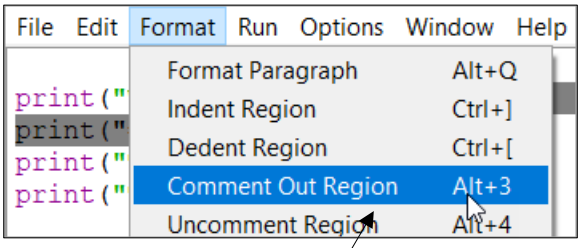
You can use three double-quotation marks in a row to mark out multiple comment lines:

```
"""
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")
```

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

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).

© Copyright 2024

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#	`// integer variable` `int answer = 42;`
SQL	`-- declare an integer variable` `DECLARE @answer int = 42`
JavaScript	`// declare a variable` `var answer = 42;`
Visual Basic	`'create integer variable` `Dim answer As Integer = 42`

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).

© Copyright 2024

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
>>>
```

© Copyright 2024

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	`**`	The following code excerpt would return 2^{10}, or 1024:
		`# base number and power`
		`base_number = 2`
		`power = 10`
		
		`# get answer`
		`answer = base_number ** power`
		
		`# show result`
		`print(answer)`

Taking the remainder or modulus of a number	`%`	The following code would return 1 (the remainder when you divide 22 by 7):
		`# base numbers`
		`start_number = 22`
		`divisor = 7`
		
		`# get answer`
		`answer = start_number % 7`
		
		`# show result`
		`print(answer)`

© Copyright 2024

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\
>>>
```

© Copyright 2024

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.*

© Copyright 2024

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.*

© Copyright 2024

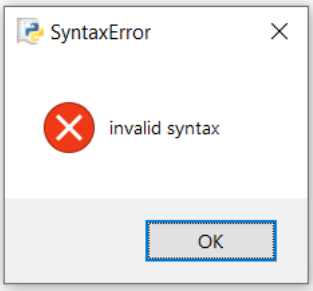
Testing for Equality

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:

```
# 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)")
```

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

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:
```

© Copyright 2024

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 and AI	Visual C#			
	VB programming			
	AI tools			
	Python			

Wise Owl Training

Wise Owl
Training