

# Practical 1 - Basic Python Programming

Welcome to practicals. Each practical is split into multiple parts:

- Part 1 is usually a walkthrough or a warm up that should not take long but shows you working code (which you can reference and learn from),
- Part 2 is usually a “fill-in-the-blanks” where you must do a small task, mostly modifying existing code,
- and Part 3 is usually a “do-from-scratch” exercise where you do the whole task with the benefit of having completed the earlier parts.
- There is also often an additional “extension” section, which we encourage you to do to practise and extend your programming skills.

**Important:** You should not expect to learn and do well in this subject if you only do the minimum contact hours (like these pracs). You need to be practising and revising your programming regularly multiple times every week. You will also get a lot of benefit from doing extra online courses like the one at:

<https://www.codecademy.com/learn/python> or the video-based ones at <https://www.lynda.com>

## Practicals are assessed.

You should aim to complete this work in the practical session, but you have one additional week to complete each of these tasks and still earn your mark. If you do not finish a practical in the scheduled week, you can still get full marks if you show it to your practical supervisor at the **start** of the next practical. After that, it's too late.

Assessment will be based on completing the tasks up to but not including the extension section to a satisfactory standard (not getting everything correct). You will be marked as follows:

- 0 – not attempted or minimal effort
- 1 – some of the work attempted with decent effort
- 2 – most/all of the work successfully completed

*Please have **all** your tasks open and ready to show your tutor when it's time to get marked.*

The online tests through LearnJCU also contribute to your practical mark. Each test is worth the same value as a prac, so your total mark is for 10 practicals + 5 tests.

See the subject outline for details of how tests are marked.

Let's start by getting you used to working with the PyCharm IDE. IDE stands for Integrated Development Environment - sophisticated software we use to write Python programs.

A lot of the code for pracs can be found at <https://github.com/CP1404/Practicals> - Many of these files have # **TODO** comments to highlight what steps to do. You may want to download or clone that repository... but we don't expect you to know how to use GitHub yet, as we will teach you that in a couple of weeks.

## Walkthrough Example - PyCharm

If you are using your own laptop, you can find complete instructions for setting up all the required software at:

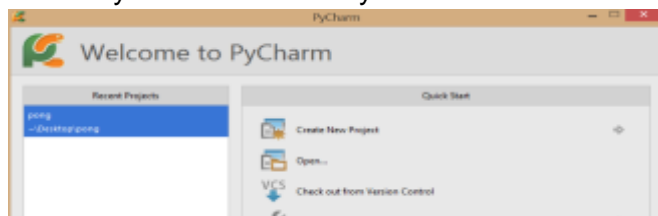
<https://github.com/CP1404/Starter/wiki/Software-Setup>

To start with you don't need Git or Kivy, just Python and PyCharm.

The first thing you need to know is how to run a Python program in PyCharm.

1. Locate and run the PyCharm software on your lab or personal computer.

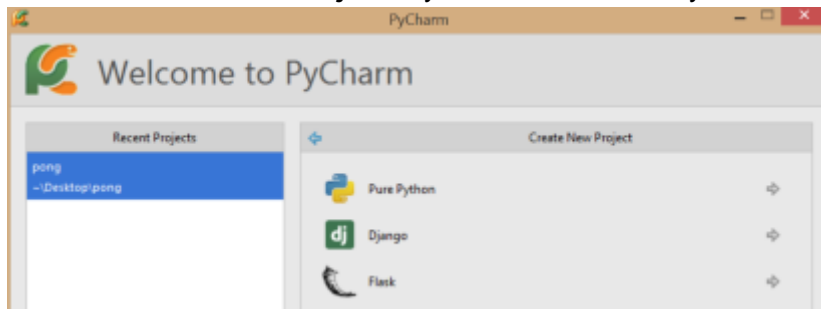
When PyCharm first starts you should see something like the following window:



Things to note:

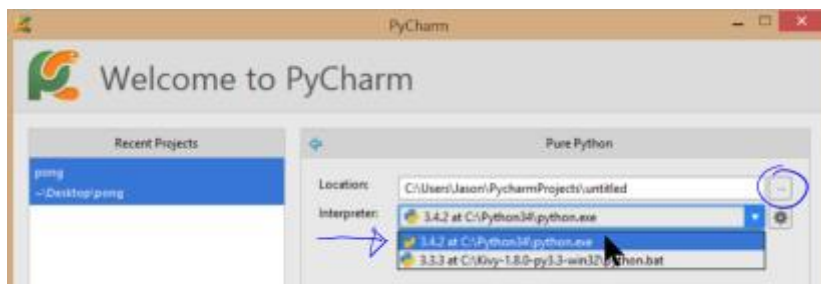
- a **PyCharm project** is a folder on the computer that contains Python source code files and related resource files to make your program run
- the **Quick Start** lists several useful tasks like creating a new project or adjusting the configuration of PyCharm

2. Click on **Create New Project**. PyCharm then allows you to choose the project type:

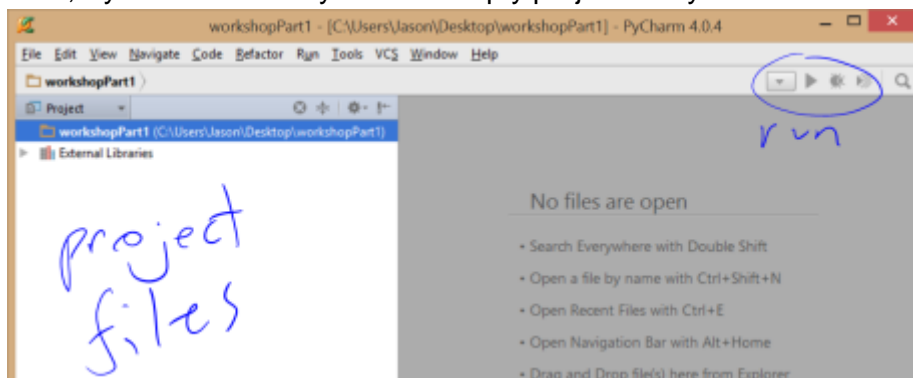


3. Click on **Pure Python**. PyCharm asks you where to store your new project and to choose an interpreter.

- the **location** can be changed to any place you have access to - if you're on a lab PC, we suggest that you work on your own USB drive
  - use the **...** button to help you select the location
- the **interpreter** is the version of Python we need to translate our Python source code into bytecode that can be run on the computer. We are using Python 3 in this subject.



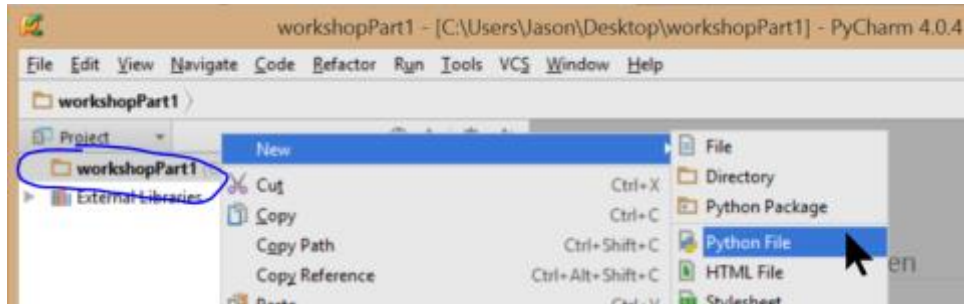
4. Next, PyCharm creates your new empty project and you should see a window like this:



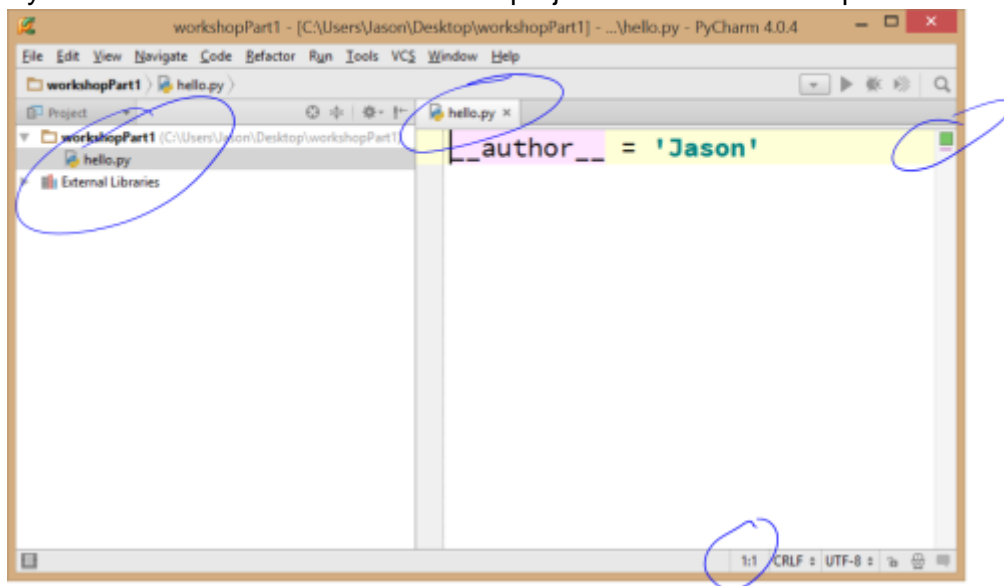
Things to note:

- the left-hand window shows the files and resources in your project
- the right-hand space is used to show a source code editor - at the moment, no source code files exist in the project
- also, there are some buttons that allow you to configure, run, and debug your program - at the moment, they are disabled
- lastly, you can see that we have called my project "practicalPart1", you can also see where it is located on the computer

5. Next, let's add some source code to the project, **right-mouse click** on the project name and select **New > Python File**



6. This opens a popup window where you can define the name of your new Python source code file and hit the **okay** button. Always give your files descriptive names so they're easy to find again.
7. PyCharm created a new text file in the project folder for us and opened the editor window to display it:



Things to note:

- the project window now shows the file (note the file extension **.py** was added automatically)
  - the green box on the right indicates there are no syntax errors at the moment (this will change :)
  - the row/column position of the cursor is displayed at the bottom of the screen, also notice how the current line is highlighted in yellow...
  - PyCharm added in one line of source code for us - the special variable **\_\_\_author\_\_\_** was defined as a string (it's defined by the current system username). Please change this to your name.
8. Let's learn our first shortcut! Press Shift+Enter to add a blank line below the one you're on (no matter where the cursor is). Nice! Do it again, then add the famous line:

```
print('hello world')
```

9. To run this standard first program, **right-click** in the code editor window and select the **run** option. (If it didn't work, please check for what the problem might be, then if you need to, ask the nearest person for help if you can't figure it out.)

In the next example, we will write a program that can compute Celsius to Fahrenheit temperature conversions.

You do not need to create new projects for each program.

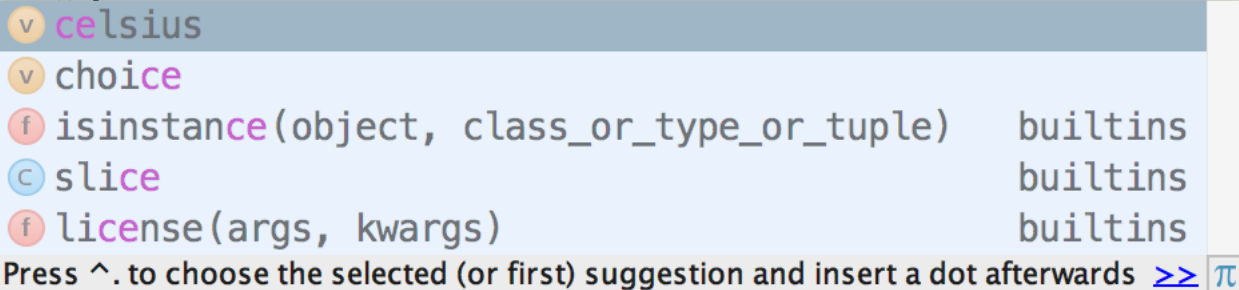
**For practicals**, use one project for all practicals, create a folder for each prac and name each of your files meaningfully.

**Do** use separate projects for each assignment and any other 'projects'.

**Do not** just use one project for everything.

1. Create a folder (directory) called **Prac01** and move your **hello.py** file into it... You can use the **Ctrl+N** shortcut to create new things, but it's context-sensitive. It creates things inside what you have selected.
2. Create a new Python file in this folder called **temperatures.py** and copy the code found at:  
<https://github.com/CP1404/Practicals/blob/master/Prac01/temperatures.py>  
This file is hosted on GitHub in our subject's practicals "repository" (or "repo" for short).  
You may want to bookmark the top level of this repo as we will be using this throughout the subject.  
Run this program (right-click in the code window and choose **run**).  
You may not understand all of this code, but you should hopefully have a good idea how it works.
3. Now see if you can complete the TODO: Replace "pass" with your code to do the opposite temperature conversion. Try it out.

```
elif choice == "F":  
    # TODO: Write this section so the program converts F to C  
    # Hint: celsius = 5 / 9 * (fahrenheit - 32)  
    cel
```



The image shows a PyCharm IDE window with a dropdown menu for the variable 'cel'. The suggestions listed are: 'celsius' (with a variable icon), 'choice' (with a variable icon), 'isinstance(object, class\_or\_type\_or\_tuple)' (with a function icon), 'slice' (with a class icon), and 'license(args, kwargs)' (with a function icon). The dropdown also includes a prompt: 'Press ^ to choose the selected (or first) suggestion and insert a dot afterwards' followed by a right arrow and a pi symbol.

Things to note:

- As you are typing code, you should notice that PyCharm is trying to help you
- As you type in the name of a variable that already exists in the code, PyCharm will show you a list of matching variable names that you can select from by simply pressing ENTER  
Don't use your mouse to click on these. Shortcuts aren't shortcuts if you use them the long way!
- The more you type in, the more specific the list becomes
- Notice also that PyCharm colours different parts of the code to help you understand what they are - variables in black, functions in blue, etc.
- You may also notice that the `# TODO` comment is special. PyCharm knows you want to come back to this. You can list all of your TODOs in a project, and it can warn you of unfinished or new TODOs when you commit your work to version control (more on that later).
- If you see an underline, move your mouse over it to see what the issue is, and look at the action item light bulb icon to see if PyCharm can automatically fix it for you!
- PyCharm offers the programmer many other useful support features... Please make use of these so you can work better and faster (it's not cheating; it's good practice!)

## Intermediate Exercises

Okay, let's practise using PyCharm to write simple programs.

1. Create a new Python file in the Prac01 directory called **salesBonus.py**, and copy the following **docstring** at the top of the file. A docstring is a triple-quoted special comment "doc(umentation) string".  
"""  
Program to calculate and display a user's bonus based on sales.  
If sales are under \$1,000, the user gets a 10% bonus.

If sales are \$1,000 or over, the bonus is 15%.  
"""

Now write the Python code to complete the program according to that docstring.  
The first line might look like:

```
sales = float(input("Enter sales: $"))
```

Run and **test** the code with a few different values to verify that it works.

Whenever you are testing code, you should not just use random values but values that you know the expected output for and that test all paths of execution, including the **boundary or edge case**.

So for this program we could use:

Test Input	Expected Output
500	50
2000	300
1000 (edge case)	150

## 2. Debugging:

Someone (it's not polite to say who) was trying to write a program to tell the user if their score is invalid, bad, passable or excellent, but their code is in the "bad" category and doesn't work.

Rewrite the following programming attempt (in a file called **fixedScore.py**) using the most efficient if-elif-else 'ladder' you can. The code is also available at:

<https://github.com/CP1404/Practicals/blob/master/Prac01/brokenScore.py>

```
score = float(input("Enter score: "))
if score < 0:
    print("Invalid score")
else:
    if score > 100:
        print("Invalid score")
    if score > 50:
        print("Passable")
    if score > 90:
        print("Excellent")
if score < 50:
    print("Bad")
```

3. Create a file called **loops.py** and add this for loop that displays all of the odd numbers between 1 and 20 with a space between each one.

```
for i in range(1, 21, 2):
    print(i, end=' ')
print()
```

Now write two more loops to do the following:

- count in 10s from 0 to 100: 0 10 20 30 40 50 60 70 80 90 100
- count down from 20 to 1: 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

4. Add a loop to the sales bonus exercise you did above, so that the program repeatedly asks for the user's sales and prints the bonus **until** they enter a negative number.  
Remember that **until** is the opposite of **while**.

## Do-from-scratch Exercises

Here are a few problems to solve "from scratch". If you need help, ask a classmate or your tutor.

### Shipping Calculator:

A shipping company requires a small program that would allow them to quickly work out total shipping charge for a number of items, each with different prices.

The program allows the user to enter the number of items and the shipping cost for each different item.

Then the program computes and displays the total shipping cost.

If the total shipping cost is over \$100, then a 10% discount is applied to the total shipping cost before the amount is displayed on the screen.

The output might look something like:

```
Number of items: 3
Price of item: 100
Price of item: 21.56
Price of item: 3
Total price for 3 items is $112.10
```

Note: start with the main logic, then adjust your program to improve the formatting.

**+ Error checking (input validation loop):** If the number of items is less than zero, a message should be displayed (e.g. "Invalid number of items!") and this quantity must be re-entered by the user **until** it is valid.

## Extension & Practice Work

Remember, these 'extension' exercises may be optional in terms of marks, but the best way to get better at programming is... programming! So do them :)

You will learn better if you don't try and do all of this in one go, but spread your work over multiple sessions.

Save each program in a different file within the Prac01 folder.

1. **Create an electricity bill estimator.** Inputs should be:
  - price per kWh in cents,
  - daily use in kWh, and
  - number of days in the billing period.

### Example use:

```
Electricity bill estimator

Enter cents per kWh: 35
Enter daily use in kWh: 4.5
Enter number of billing days: 90

Estimated bill: $141.75
```

2. Modify your bill estimator by asking the user to choose which tariff they are using - then use the appropriate stored value for cents per kWh.  
Start by defining two **constants** like below.  
Constants in Python are just variables written in ALL\_CAPITALS.

```
TARIFF_11 = 0.244618
```

TARIFF\_31 = 0.136928

**Example use:**

Electricity bill estimator 2.0

Which tariff? 11 or 31: 11

Enter daily use in kWh: 13.4

Enter number of billing days: 90

Estimated bill: \$295.01

3. **Menus:**

One very common programming task is to make menus by combining looping (repeat the program until the user quits) with selection (let the user decide what to do).

The general pattern of a menu-driven program is as follows:

```
display menu
get choice
while choice != <quit option>
    if choice == <first option>
        <do first task>
    else if choice == <second option>
        <do second task>
    ...
    else if choice == <n-th option>
        <do n-th task>
else
    display invalid input error message
display menu
get choice
<do final thing, if needed>
```

Note that a common error when writing menus is to forget to repeat the menu display and prompt at the end (inside) the loop.

Use this pattern to create a very simple menu-driven program according to the pseudocode below:

```
get name
display menu
get choice
while choice != Q
    if choice == H
        display "hello" name
    else if choice == G
        display "goodbye" name
    else
        display invalid message
display menu
get choice
display finished message
```





Sample output for this program should look like (green text represents user input):

```
Enter name: Guido
(H)ello
(G)oodbye
(Q)uit
```

```
>>> A
Invalid choice
(H)ello
(G)oodbye
(Q)uit
```

```
>>> H
Hello Guido
(H)ello
(G)oodbye
(Q)uit
```

```
>>> G
Goodbye Guido
(H)ello
(G)oodbye
(Q)uit
```

```
>>> Q
Finished.
```

#### 4. Menu-driven number sequence generator:

A school teacher requires a small program that would allow primary school students to learn about various number sequences. The teacher is interested in a simple menu-driven program that has the following choices (where x and y are inputs the user enters once at the start of the program):

1. Show the even numbers from x to y
2. Show the odd numbers from x to y
3. Show the squares from x to y
4. Exit the program

## Answers to loop example questions

```
for i in range(1, 21, 2):
    print(i, end=' ')
print()
```

```
for i in range(0, 101, 10):
    print(i, end=' ')
print()
```

```
for i in range(20, 0, -1):
    print(i, end=' ')
print()
```