





Branch: master ▾ Practicals / prac\_01 /

Create new file Find file History

 lindsaymarkward	Reuse same (repeated) images, remove now unused images	Latest commit 282c60d 7 days ago
..		
 README.md	Reuse same (repeated) images, remove now unused images	7 days ago
 broken_score.py	Update folder and files based on pylint style, and various other updates	a year ago
 temperatures.py	Update folder and files based on pylint style, and various other updates	a year ago

 README.md

## Practical 01 - Basic Python Programming in PyCharm

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.
- The practice/extension section provides further challenges for you to practise and extend your programming skills. (You do not need to complete these sections for your prac marks.)

**Important:** You should not expect to learn and do well in this subject if you only do the minimum contact hours (like just these pracs). You need to be practising and revising your programming regularly multiple times every week. You can find even more practice projects at <https://github.com/CP1404/Starter/wiki/Practice-Programming-Projects>

Some students have found benefit from doing extra online courses like the video-based ones at <https://www.lynda.com>

### Practicals are assessed

You should aim to complete the 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 practice/extension section to a satisfactory standard (not getting everything correct). You will be marked as follows:

0. not attempted at all
1. some of the work attempted with minimal effort
2. some of the work attempted with reasonable effort
3. most or all of the work successfully attempted

Note that to get full marks, you must have attempted every part of every question. Make sure you read the questions carefully, and do everything required.

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

There are online tests available through LearnJCU that 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. We recommend doing each test before you start your practical work, and you must complete it by the due date (the week it's in) to get your marks for it.

All code for pracs can be found here 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 the practicals 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

Let's start by getting you used to working with the PyCharm IDE (Integrated Development Environment).

If you are using your own laptop, complete the 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 3 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 have a window with a link to create a new project.

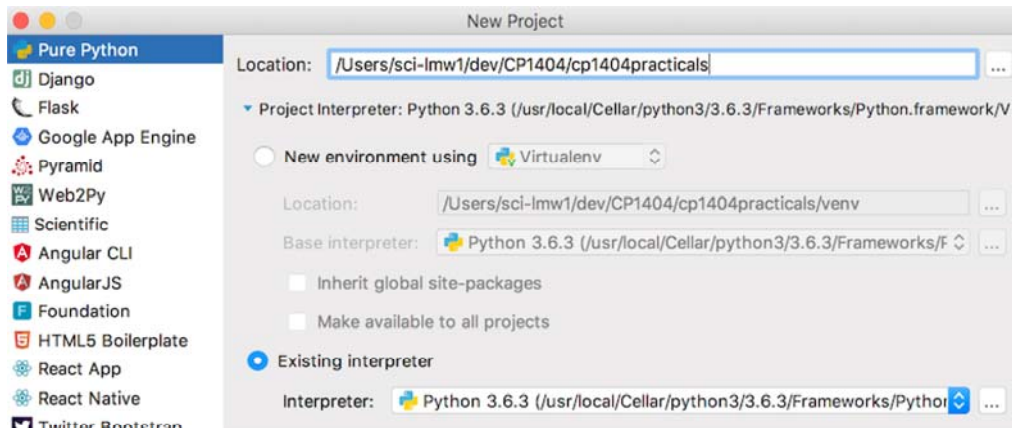
- A PyCharm **project** is a folder on the computer that contains Python source code files and related resource files to make your program run... but it's more than that. Always do your coding in files that are part of a meaningful project.
- the **Quick Start** window lists several useful tasks like creating a new project or adjusting the configuration of PyCharm

2. Click on **Create New Project**, and choose **Pure Python**. PyCharm asks you where to store your new project and to choose an interpreter. Your screen may look different than our image below, but name your new project (folder) for your **cp1404practicals**.

- the **location** can be changed to any place you have access to. Use a folder that you will be able to find later
- use the ... **button** to select the location
- the **interpreter** is the version of Python we need to run our code on the computer.

Use the *existing interpreter* for Python 3.

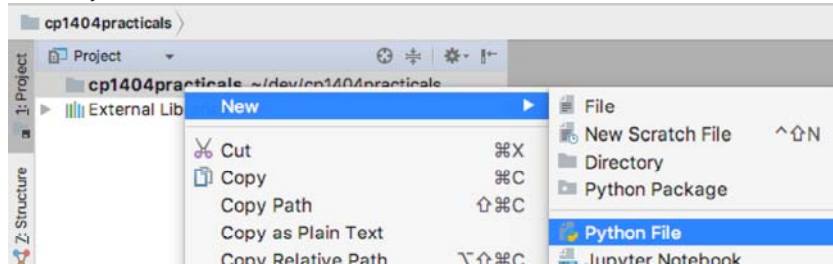
**DO NOT** use a virtual environment (venv). They're cool but we don't need them in this subject.



- Next, let's add our first source code file to the project.

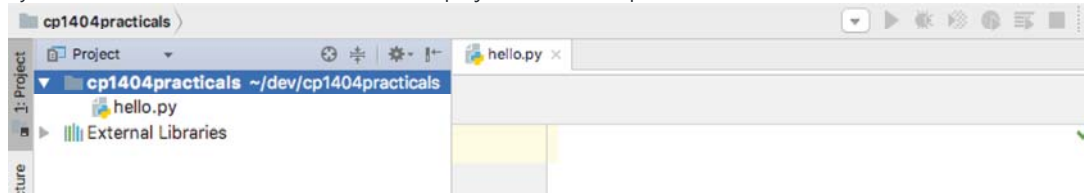
Right-click on the project name and select

New > Python File



- This opens a window where you can define the name of the file - call it **hello.py** and hit the **OK** button. Always give your files descriptive names so they're easy to find again.

- PyCharm created a new source code file in the project folder and opened the editor window:



Note:

- the project window now shows the file (with the file extension **.py**)
  - the green box/tick up the top right indicates there are no syntax errors... yet :)
  - 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 may also have added in one line of source code for us - the special variable `__author__` (defined by the current system username). This doesn't matter much.
- 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')
```

- 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.)

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

## Project Structure: Use one project for all practicals

Your project name and structure should look like our template: <https://github.com/CP1404/cp1404practicals>

So you will have one well-named folder for each prac (names like "prac\_01" conform with Python module name conventions), with well-named files (also use underscore\_lowercase) inside those folders.

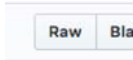
**Do not** create new projects for each separate practical.

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

- If you don't have your first prac folder (directory) yet, create one called **prac\_01** and drag your **hello.py** file into it...

2. Right-click on your `prac_01` folder and create a new Python file here called `temperatures.py`. Copy the raw version of the code found at the link below and paste into your file...

Click on the link below, then click the Raw button to get a version suitable for copying.



Don't just copy from the normal view without clicking Raw, or you will *not* get proper formatting. OK?

Download [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.

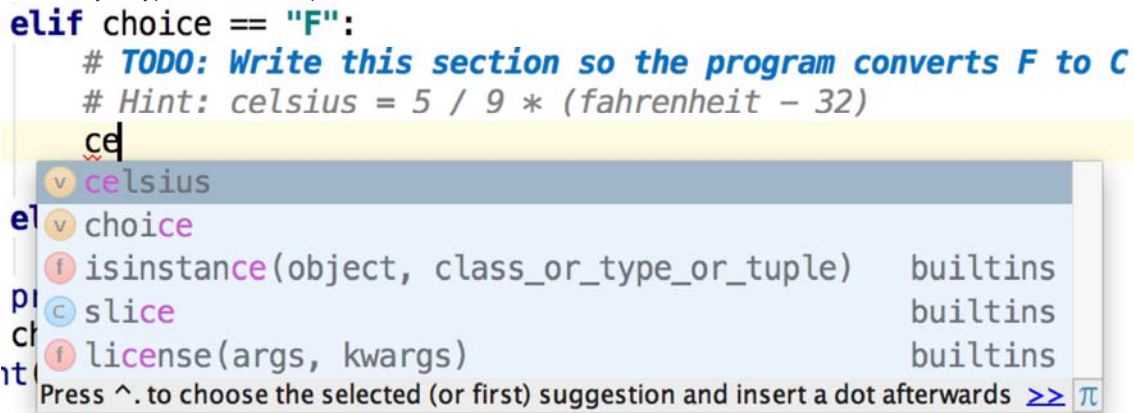
3. **Run** this program by right-clicking in the code window and choosing **run**. Try converting a couple of Celsius values to Fahrenheit.  
You may not understand all of this code, but you should hopefully have a good idea how it works.
4. Now see if you can complete the TODO. Replace "pass" with your code to do the opposite temperature conversion (F to C). Try it out.

## Productivity Tips

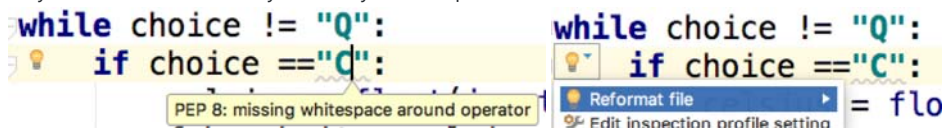
Good programmers (that's you!) don't waste time. Here are some important tips to save you time...

- As you are typing code, you should notice that PyCharm is trying to help you with suggestions and highlights.
- As you type in the name of an existing identifier, 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



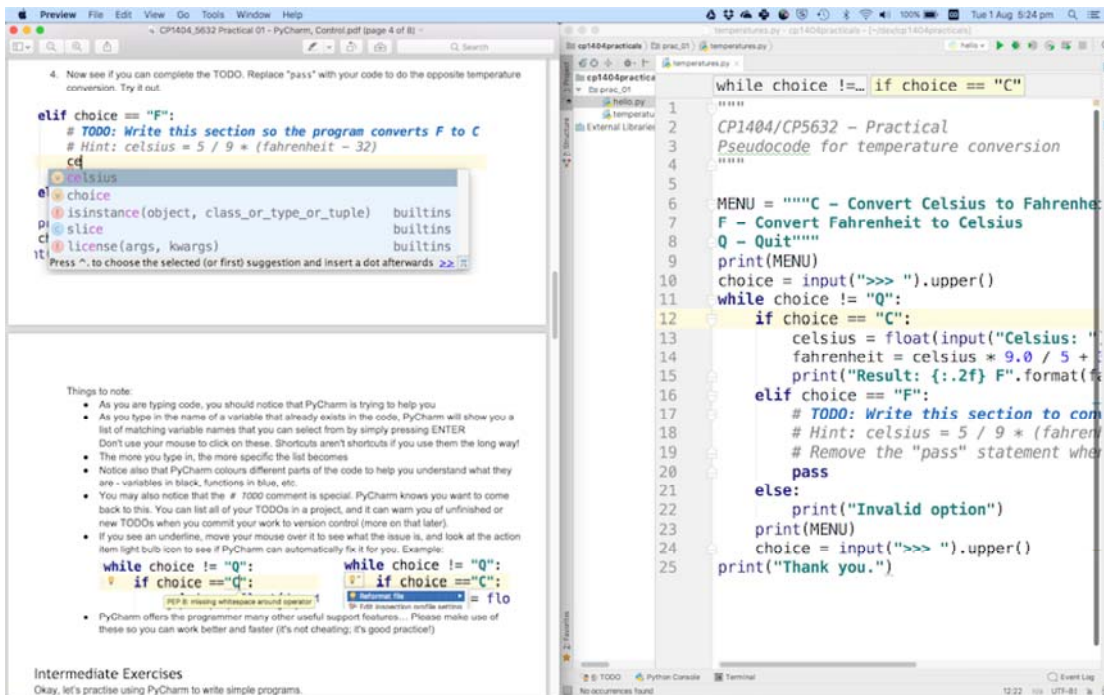
- Notice also that PyCharm colours different parts of the code to help you understand what they are - variables in black, functions in blue, etc.
- You may also notice that the `# TODO` comment is special. PyCharm knows you want to come back to this. You can list all of your TODOs in a project, and it 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. Example:



- 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!)

While we're thinking about being more productive...

A good way to organise your workspace is to put these instructions on one side of your screen, and your code (PyCharm) on the other side. (Windows users can use the Windows key + arrow shortcuts for this.) Now you don't have to waste time switching back and forth between these two screens! Drag the bar between your project files and your code so you can see lots of code (or hide the project window altogether with Alt+1).



Got your windows organised? Let's keep going...

## Intermediate Exercises

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

1. Create a new Python file in the `prac_01` directory called `sales_bonus.py`, and copy the following **docstring** at the top of the file. A docstring is a triple-quoted special comment "documentation" 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 the following (we're only interested in the values, not the format):

Test Input	Expected Output
500	50

Test Input	Expected Output
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 using the most efficient if-elif-else 'ladder' you can. The code is available here at: [broken\\_score.py](#)

Remember to click **Raw** before copying and pasting so you get proper formatting!

The *intention* is that the score must be between 0 and 100 inclusive; 90 or more is excellent; 50 or more is a pass; below 50 is bad.

Be very careful of your boundary conditions... and *test*!

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 more for loops** (using range) to do the following:

a. count in 10s from 0 to 100: 0 10 20 30 40 50 60 70 80 90 100

b. 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

c. print n stars. Ask the user for a number, then print that many stars (\*), all on one line

*Note: this is a very simple loop for repeating n times. We use for loops for "definite" iteration like this. while loops are used for "indefinite" iteration (like repeating while a user input is incorrect).*

d. print n lines of increasing stars. Using the same number as above print lines of increasing stars, starting at 1. E.g. if 4 was the number entered, your single loop should print

```
*
**
***
****
```

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**.

*Keep going...*

We've now had the walkthrough and the intermediate exercises where you mostly have existing code to modify and extend, or you have to write something really similar to working code you've been given...

Now we move on to the section where you are given a task/goal to complete with no starting code. Remember to use the examples done above as a guide, and if you're not sure how to do it, go back to the subject materials. For example, the question below asks for an error-checking loop, which we cover in chapter 2 (Control), and which is also summarised as one of our standard patterns at: <https://github.com/CP1404/Starter/wiki/Programming-Patterns>

## Do-from-scratch Exercises

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

## Shop Calculator

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

The program allows the user to enter the number of items and the price of each different item.

Then the program computes and displays the total price of those items.

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

The output should look something like (**bold text** represents user input):

```
Number of items: 3
Price of item: 100
Price of item: 35.56
Price of item: 3.24
Total price for 3 items is $124.92
```

Create the file `shop_calculator.py` and write this program.

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

Note that the example output above uses string formatting to set the currency to 2 decimal places... don't worry about this for now... or do :)

+ **Error checking (input validation loop):**

(Do this *after* you have completed the above program.) If the number of items is less than zero, the message "Invalid number of items!" should be displayed and this quantity must be re-entered by the user *until* it is valid.

## Practice & Extension Work

Remember, even though these 'extension & practice' exercises are optional in terms of marks, but the best way to get better at programming is... *programming!*

So do them each and every week :)

You will learn better if you spread your work over multiple sessions instead of trying to do all of this in one go.

Save each program in a different file within the `prac_01` 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:

```
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):

- i. Show the even numbers from x to y
- ii. Show the odd numbers from x to y
- iii. Show the squares from x to y
- iv. Exit the program

## Solutions?

---

Remember that solutions are provided for most prac exercises... to help you learn, not just for copying :)  
See the solutions branch here in this repository. Change the **branch** to "solutions" at the top of the page.