1. Variable naming conventions

Below are some tips for ensuring your variable names follow naming conventions for Python.

- Variable names must be **meaningful**. They should clearly indicate what the variable is for (ie the data it holds).

Example

You want to create a variable to hold the name of a superhero.

Variable names:

✓ `superhero` or `superhero_name` would be fine

✗ `SH`, `sh`, `SHN`, `sup_hero`, etc could be confusing. If you looked at the code in six months' time, you may have forgotten what these are for.

- For variables with more than one word, use:
  - **lowercase with underscore** (eg `user_score`) – this is the style preferred by most Python programmers <u>**OR**</u>
  - **camelCase** (eg `userScore`)

  Never use both styles together (eg `first_UserScore`).

- Keep words in a variable name to a minimum.

  ✓ `birth_date` or `birthDate`

  ✗ `date_on_which_user_was_born` or `dateOnWhichUserWasBorn`

2. Following conventions

In addition to naming conventions, it is also important to adhere to conventions around:

- maximum line length
- commenting.

Maximum line length conventions

The convention: All lines of code should be no more than 79 characters long. Multiline comments should be even shorter (see next section on **Commenting conventions**).

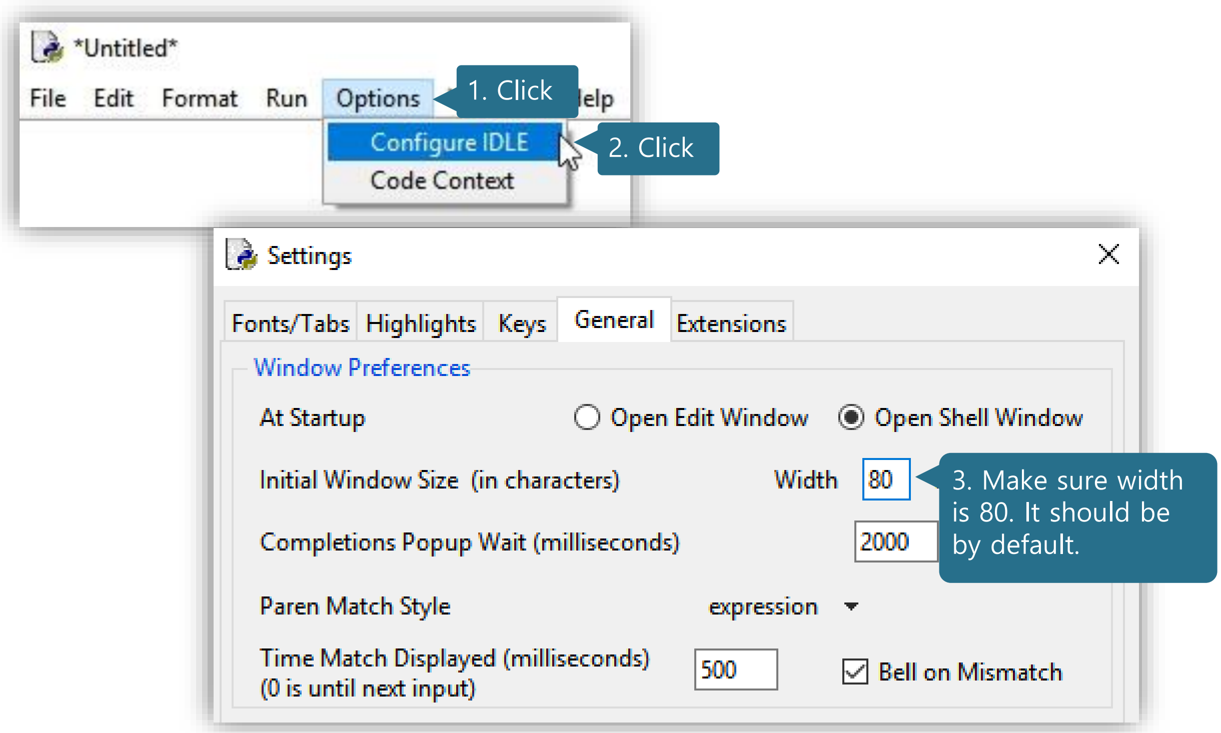
Reason for convention: Allows you to see all the code using a small window size, so you can have several files open next to each other. This is useful when you need to:

- compare code in files
- copy and paste code from one file to another, etc.

How to do it:

In IDLE

- Make sure your window size in IDLE is set to a maximum width of 80 characters. To do this:



Window Preferences

At Startup ( ) Open Edit Window (●) Open Shell Window

Initial Window Size (in characters) Width 80

Completions Popup Wait (milliseconds) 2000

Paren Match Style expression

Time Match Displayed (milliseconds) (0 is until next input) 500 ☑ Bell on Mismatch

Continues next page

When you open a new IDLE window, the width of the window will be set to 80 characters. Do **not** change the window size. This will allow you to easily see where the 79 character maximum line length is.

- If you have a line of code longer than 79 characters, you cannot just press Enter on your keyboard to move to a new line. This will cause an error.
- Instead use a **back slash**. Example:

```
age = 4
room = 6
if age > 2 or age < 6 and room == 6 or room == 13 or age > 6 and age < 12 and \
room == 6 or room == 12:
    print("Yes")
```

Type a space, then \, and press Enter.

Note: For a function that includes ‘+’ or ‘,’ in the argument, you can press Enter on your keyboard to move to a new line. Look at the examples below.

```
easygui.msgbox("Hi " + player_name +
                 "! Welcome to the Guessing Game.")

print("This print line includes text (string) and variables",
       variable_1, variable_2)
```

You can press Enter on your keyboard here

You can press Enter on your keyboard here

In Visual Studio Code

1. Open any file in Visual Studio Code.
2. Resize the window so that it is a maximum of 79 characters wide (and whatever height you want it to be).
3. On the menu select **File > Preferences > Settings**.
4. In the search bar type: **window.newWindowDimensions**.
5. Select the value **inherit** from the dropdown box.

Going forward, every time you open Visual Studio Code, the size of the window will be based on the size of the last window you had open.

Commenting conventions

In addition to the basics covered in your workbook, the following are important conventions for commenting.

Inline comments

- Leave at least two spaces between the code statement and the comment.
- Start the comment with a # followed by a space.

Compare the code blocks that follow:

Correct commenting

```
while age <5 or age > 16:  # Checks user has entered valid age range.
      age = int(easygui.enterbox("Please enter valid age: "))
```

Incorrect commenting

```
while age <5 or age > 16:#Checks user has entered valid age range.
      age = int(easygui.enterbox("Please enter valid age: "))
```

Comments for blocks of code

For a block of code, add the comment at the start of the block, not inline with the first line of code. The comment should be indented to the same level as the code.

Correct commenting

```
age = int(easygui.enterbox("How old are you?"))
      # Checks user has input valid age range.
      while age <5 or age > 16:
            age = int(easygui.enterbox("Please enter valid age: "))
```

Continues next page

Incorrect commenting

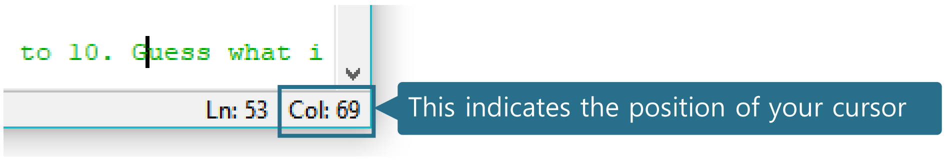
```
age = int(easygui.enterbox("How old are you?"))
# Checks user has input valid age range.
      while age <5 or age > 16:
            age = int(easygui.enterbox("Please enter valid age: "))
      while age <5 or age > 16:
            age = int(easygui.enterbox("Please enter valid age: "))
```

Line length for block comments

Block comments need to be a maximum of 72 characters long. This is so that they stand out clearly from the rest of the code. Example:

```
while age <5 or age > 16:
      # This is a multiline comment. It stands out from the rest
      # of the code because it has a shorter line length.
      #
      # To indicate a new paragraph, use a single # on the line (as
      # shown above.
      age = int(easygui.enterbox("Please enter valid age from 5 to 16: "))
```

To check how long your comment is, look at the bottom right of your window.



```
to 10. Guess what i
Ln: 53  Col: 69
```

Write comments for humans!

Remember that people read the comments, not the computer. Make sure your comments are:

- full sentences that start with a capital letter (unless the sentence starts with an identifier such as a variable name)
- in English - only write them in another language if you are completely sure anyone who might read them can understand that language
- written in a way anyone could understand.

Continues next page

The comment below is **not** suitable.

```
#user input = valid age rng?
```

It is not a proper sentence and uses an abbreviation (rng) that some people may not understand.

Ensuring comments are meaningful and useful

Comments need to describe the **function** and **behaviour** of the code.

Do not include comments if this information is already clear from the code itself.

Compare below.

Unuseful comment

```
round (answer) # Round the answer.
```

Comment is not useful because this is already clear from the code.

Useful comment

```
round (answer) # Answer rounded to allow for margin of error.
```

Comment indicates behaviour of code (rounding) and why it has been included (function).

3. Effective testing and debugging

To test your program effectively and efficiently, you need to do the following.

Test and debug:

- in an organised way
- using **expected** and relevant **boundary** cases.

Organised testing and debugging

A test plan is one way to ensure your testing and debugging is organised.

The test plan should include:

- what you will be testing and how
- what the expected result is
- whether you got the expected results, and if not …
- what you did to correct it.

An example is shown below – however, you could use a different format.

Code to be tested:

The program below is meant to check if the user is 12 years or older. If they are, they go to Group A; otherwise they go to Group B.

```
age = int(input("Enter an age from 5 to 16: "))
if age > 12:
        print("You are in Group A")
else:
        print("You are in Group B")
```

Test Plan (partial)

Test (enter)	Output expected	Correct?	Notes
age = 6	"You are in Group B"	✓	
age = 14	"You are in Group A"	✓	
age = 7	"You are in Group B"	✓	

age = 12	"You are in Group A"	✗	*Output was 'Group B'. Checked code. Incorrect operator used in line 2. Should be:*
			if age >= 12 OR
			if age > 11.

Continues next page

- The first three tests on the test plan are examples of **expected** cases. This is input you expect (ie a number from 5 to 16).

- Test 4 on the plan is an example of a **boundary** case. A boundary case is one at the limit of a particular case. In the example shown, an age of 12, would sit on the boundary between what would be Group A, and what would be Group B.

  Errors often occur at these boundaries, so it is important to check these values.

  For the example shown, you could test a value:

  - at the boundary (ie 12)
  - just below the boundary (ie 11)
  - just above the boundary (ie 13).

- A note has been made of the error identified and what was done to fix it.

Organised and systematic testing and debugging

To ensure your testing is **organised**, make sure you systematically check each section of code where an error could occur.

- Decide on logical 'blocks' of code. For example, look at how the code for the Snake Game is broken down into different 'components' at the end of Section 2.4 in the Learner Workbook.
- Check each section of code carefully before moving on to testing the next section.

Don't wait until you have completed the whole program, or a big chunk of the program! This makes it very difficult to tell where the error is, and to debug.

Excellence

For the **step up to Excellence** you need to also:

- test and debug **invalid** cases (not only expected and boundary cases)
- check the user input for validity.

To learn how to do this, work through the next two sections on:

- comprehensive testing and debugging
- checking input validity.

4.Comprehensive testing and debugging

To comprehensively test and debug your code, you need to test expected, boundary, and invalid input.

- You've already learnt about expected and boundary cases.
- Invalid cases are inputs that are clearly and obviously wrong. Using the code shown below, an invalid case could be if the user:
  - does not enter anything for their name and/or for their age
  - enters an age under 5 or above 16 (they should enter age from 5 to 16)
  - presses the Cancel button (rather than entering an answer).

```
import easygui

name = easygui.enterbox("Hi! What is your name?")
age = easygui.integerbox(str(name) + ", enter an age from 5 to 16: ")
if age >= 12:
     easygui.msgbox("You are in Group A")
else:
     easygui.msgbox ("You are in Group B")
```

Extension Activity B: Comprehensive testing and debugging

Try testing the code above using invalid inputs. (Name the file: **age_group_v1**)

What happens if you:

- don't enter a name when prompted (ie just press the Enter key)
- enter an invalid age
- press the Cancel button rather than entering your name?

5. Checking input validity

Checking the user's input to make sure it is valid, is called **input validation**.

We are going to look at three key types of input validation.

- Presence check – checks that a value has been input
- Range check – checks that a value is in a reasonable range
- Cancel button check – checks for if the Cancel button has been pressed

We will also look at how to:

- convert a lowercase string to uppercase. This could be useful, for example, if you ask the user to input "Y" for "yes", and they enter "y" instead
- capitalize the first letter of a string. This could be useful, for example, if the user may input a string with or without a capital letter.

Presence check

If the user does not input anything when prompted, the variable will be set as `""` (ie an empty string).

To evaluate if there is no input, use a `while` loop as shown below.

```
1  import easygui
2  name = ""
3  while name == "":
4        name = easygui.enterbox("Hi! What is your name?")
5  age = easygui.integerbox(str(name) + ", enter an age from 5 to
   16:  ")
6  if age >= 12:
7        easygui.msgbox("You are in Group A")
8  else:
9        easygui.msgbox ("You are in Group B")
```

Notes:

- **Line 2** sets name as an empty string – so that the `while` loop (starting in **line 3**) will execute at least once.
- **Line 3** checks whether `name` is an empty string. If it is, **line 4** executes. Otherwise, line 4 is skipped and program moves to **line 5**.

Range check

Below is an example of how a range check (highlighted code) could be added to the code on the previous page.

```
import easygui
name = ""
while name == "":
      name = easygui.enterbox("Hi! What is your name?")
age = easygui.integerbox(str(name) + ", enter an age from 5 to
16: ")
while age <5 or age > 16:
    age = easygui.integerbox("Please enter a valid age from 5 to
16: ")
if age >= 12:
      easygui.msgbox("You are in Group A")
else:
      easygui.msgbox ("You are in Group B")
```

The `while` loop checks that the user has input an age within the valid range.

- If they have **not**, they are asked to input a valid age.
- If they have, the error message is skipped, and the rest of the code runs.

Cancel button check

For a program that uses EasyGui, if the user presses the **Cancel** button, the value `None` is returned.

- `None` is used to define a null value (ie no value). It is **not** the same as 0.
- `None` is its own datatype – NoneType.

To follow is an example of how to handle if the user presses the Cancel button when prompted to enter their name.

Continues next page

```
import easygui
name = ""
while name == "":
    name = easygui.enterbox("Hi! What is your name?")
    if name == None:
        leave = easygui.buttonbox("Do you want to exit?",
         choices=["Yes", "No"])
        if leave == "Yes":
            quit()
        else:
            name = ""
```

Converting lowercase to uppercase

Look at the example below. Because Python is case sensitive, if the user entered `y` rather than `Y` the else statement would execute.

```
play = input("Do you want to keep playing? Y/N  ")
if play == "Y":
    print("Welcome to the game.")
else:
    print("Goodbye")
```

You can get around this by using `if play == "Y" or "y":`

However, a more efficient way to do this would be to convert the user's input to uppercase. Then, regardless of whether they enter a lowercase or uppercase 'y', it will be treated as uppercase.

Continues next page

The `upper()` method is used for this.

```
play = input("Do you want to keep playing? Y/N  ")
if play.upper() == "Y":
    print("Welcome to the game.")
else:
    print("Goodbye")
```

Note: In a similar way, the `lower()` method can be used to convert to lowercase.

Capitalising the first letter of a word

Sometimes when you ask the user to input a string, they may enter it starting with a capital letter, or starting with a small letter. Because Python is case sensitive, this can cause problems. To get around this issue, you can use the `capitalize` method to automatically capitalise all strings.

Look at the example below.

```
name = input("Enter a name for the superhero:  ")
if name.capitalize() == "Supercat":
    print("Welcome superhero.")
else:
    print("You're no superhero!")
```

Because the string variable (name) has been capitalised, even if the user inputs 'supercat' rather than 'Supercat', they should still see the message: 'Welcome superhero.'

6. Literals, variables, derived values, and constants

For Excellence, you need to make sure your program is flexible and robust.

flexible: it should be easy to make changes to the program so that it can be used in different contexts, etc.

robust: the program should be able to cope with incorrect input from the user. See the section on Efficient testing and debugging (Checking input validity) for how this can be done.

One way to do this is to use variables, derived values, and constants rather than literals.

Literals

A literal is when you hard-code in a value. For example, in the code below, 18 is a literal value.

```
if user_age < 18:
```

This is a literal

Wherever possible use variables, constants, or derived values rather than literals.

Variables

Below is an example of where using variables could be better than using literals.

Sometimes what is inside the brackets for an EasyGui function can be very long and confusing. Look at the example below:

```
import easygui
easygui.buttonbox(msg="Which of the following is most important?",
title="What's important?", choices=["Enough to eat", "A nice house",
"Going overseas"])
```

Continues next page

Instead you could set the variables first and then get `easygui.buttonbox` to use them as follows.

```
import easygui
msg = "Which of the following is most important?"
title = "What's important?"
choices = ["Enough to eat", "A nice house", "Going overseas"]
easygui.buttonbox(msg, title, choices)
```

Variables set

EasyGui told to use the variables

This makes the code a lot cleaner and easier to understand. It also makes it easier if you want to change something (eg the message, title, etc).

Using variables from a list

Using variables outside the EasyGui functions also works with lists. Look at the example below. In this example there are two lists: `choice_questions[]` and `possible_answers[]`.

```
choice_questions = ["If you have four apples and you give two away, how \
many will be left?",
             "You start with one friend and then get three more. How many \
friends do you have?"]
possible_answers = [("2", "3", "4", "5"),
                    ("2","7","9","4")]

for i in range(0, len(choice_questions)):

        msg = choice_questions[i]
        title = "Question " + str(i+1)
        choices = possible_answers[i]

        player_answer = easygui.buttonbox(msg, title, choices)
```

The **first** time the `for` loop runs, it sets the following for the EasyGui buttonbox:

- The message (`msg`) - as the first item in the `choice_questions` list `[0]`.
- The title - as the string 'Question' followed by the number of the question – determined using the list index `[0]` and adding 1.

Continues next page

- The choices for the buttons – as the first item in the `possible_answers` list (ie `("2", "3", "4", "5")`).

The **second** time the loop runs, it sets the:

- message as the second item in the `choice_questions` list `[0]`
- the title using a question number determined using the list index `[1]` + 1.
- Choices for the button as the second item in the `possible_answers` list (ie `("2", "3", "4", "5")`).

If the `choice_questions` list had more than two items, the loop would continue in the same way for each of the questions in the list.

Derived values

Derived values are also called **calculated values**. These are values that you get as a result of an expression.

Example:

You have a program that gets the ages of all the children in a class. You need a variable to hold the average age of the children in a class. Rather than doing the calculation yourself, get the program to do it.

```
class_average_age = age_total/children
```

The advantages of using a derived value is that it:

- saves time
- avoids human error.

Constants

A constant is a variable that does not change throughout the program. Compare the two bits of code below. The first uses a literal; the second uses constants.

```python
user_age = int(input("How old are you?"))
if user_age < 18:
    print("Sorry, you aren't old enough to buy this product.")
else:
    print("You're old enough to buy this product.")
```

This is a literal

```python
LEGAL_AGE = 18
user_age = int(input("How old are you?"))
if user_age < LEGAL_AGE:
    print("Sorry, you aren't old enough to buy this product.")
else:
    print("You're old enough to buy this product.")
```

This is a constant

This is a constant

Both bits of code do the same thing: Check to see if a user is legally old enough to buy alcohol and give them an appropriate message.

- The difference:
    - The first code snippet uses a literal to check whether the user's age is below 18.
    - The second code snippet uses a constant (LEGAL_AGE). The program checks whether the user's age is less than the value of the LEGAL_AGE constant.

How does using a constant make your code more flexible?

- Imagine the legal age for buying alcohol changes to 21.
- If you use a literal, you'd have to find all the places in your code where you have checked for 18 and change it to 21.
- By using a constant, you only need to change the constant's value to 21.

Continues next page

Conventions for naming constants

- Most of the conventions for naming variables also apply for constants. The one difference is that constants are in all caps.

  Example: `LEGAL_AGE`, not `legal_age`

- Put the constants at the beginning of the program (after any `import` lines) or at the start of the block of code where it is used. This makes it easier to find if you need to change it later.

7. Effective coding

For Excellence, you need to ensure that your program uses actions, conditions, control structures and functions effectively. In this section, we look at two ways in which to do this:

- passing values from one function to another
- creating a menu using a switch statement and a dictionary.

Passing values from one function to another

Values can also be passed from one function to another. However, you must make sure the function is defined before it is called.

Look at the extract below:

```
def query_cancel():
     """Input validation: If the user presses the Cancel
     button, they are asked if they want to exit.
     """
     leave = easygui.buttonbox("Do you want to exit?",
     choices=["Yes", "No"])
     if leave == "Yes":
        quit()
     else:
        return 0
```

This value is to return to where the function was called (ie within the `validate_name()` function: `school_name = query_cancel()`).

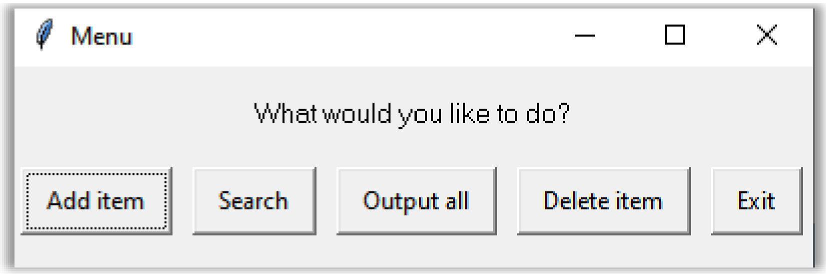
```
def validate_name()
    """Input validation: Checks that the user has input a   valid
    school name.
    """
    if school_name == None:
       school_name = query_cancel()
```

The `query_cancel()` function is called here.

```
    while school_name == "" or school_name == 0:
       school_name = easygui.enterbox("A school name must be
       entered.", title="Error")
```

Creating a menu using a switch statement

A common feature of many programs is a menu from which the user can choose an option. For example, at the start of a program, the user might see a menu that looks like this:



This menu could be created with a series of `if … else if …` statements. However, a more effective way to do this is to use a switch statement and a dictionary. Look at the example below.

```
def display_menu():
    '''Displays menu and gets user to select item from menu'''
    menu = {
        "Add item": add,
        "Search": search,
        "Output all": ouput_items,
        "Delete item": delete,
        "Exit": leave,
        }

    get_input = "Y"
    while get_input != "N":

        msg = "What would you like to do?"
        title =  "Menu"
        choices = []
        for i in menu:
            choices.append(i)

        selection = easygui.buttonbox(msg, title, choices)

        get_input = menu[selection]()
```

Dictionary with text to go on each menu button (key), and name of function (value).

Rather than hard-coding the choices list (buttons), a `for` loop is used to create the choices list from the dictionary.

This is the switch statement. See the notes below for what it does.

The switch statement calls a specific function, based on the user's selection from the menu. For example, if the user clicks on the **Delete item** button, it will be `get_input = menu[delete]()`, so the `delete()` function will be called.

8. Using classes and objects

Another way to make your code as flexible and robust as possible is to apply an object-oriented programming (OOP) approach. OOP is applied using **classes** and **objects**.

Classes and objects provide a way to:

- reuse blocks of code
- keep related code together – which ensures a clearer structure for your program
- allow for easy creating of multiple objects, based on a specified class (see below for more on this).

A **class** is like the template or blueprint for an object. For example, for the snake game, you might create a 'food' **class**, on which you can base the creating of one or more **objects** of food for the snake to eat.

Below is an example and some tips for how to do this for the snake game. **Note**: You could add this in the main program file, or create a separate file(s) for your class(es).

In this example, we are going to create a separate file called `food.py` to create the `Food` class. We will use the `Food` class to create food objects (eg an apple).

In the `food.py` file ...

Include all relevant modules and variables – as shown below.

```
# Import modules for program.
import pygame, random

MOVE_BLOCK = 20
```

Note: You can put all modules to be imported into a single statement as shown here.

Next let's look at how to work with classes and objects.

Continues next page

To create a class

```
class [Name of class]:
```

Example

Add the following in your `food.py` file.

```
class Food(pygame.sprite.Sprite):
```

Notes:

- The name of the class should start with a capital letter – to distinguish it from any variables.
- **Pygame sprite module**

  The code in brackets is calling the "Sprite" class in Pygame sprite module. This allows the new class you have just created to be a Pygame sprite.

  For more on the Pygame Sprite module see the Pygame documentation at:

  https://www.pygame.org/docs/tut/SpriteIntro.html and

  https://www.pygame.org/docs/ref/sprite.html

  or go to https://www.pygame.org/docs/ and do a search using the keyword **sprite**.

  In particular, you are likely to find the following useful for 2D games:

  `pygame.sprite.Group()` – To group several sprites together (ie creates a 'container' for several sprites.

  `pygame.spritecollide()` – Used for collision detection for sprites.

To initialise an object:

```
def __init__(self):
```

- Every class needs the above **method** which tells it **how** to create (**initialise**) an object.

Note: A **method** is like a function except it:

- is called on an object
- can use all the data inside the object and any parameters passed to it.

Continues next page

- This is a special type of method called a **magic (dunder) method**.
- It always has a double underscore before and after `init`.

Note: To do this using the Sprite class, use the following:

```
def __init__(self):
    super().__init__()
```

Allows you to call methods from the `Sprite` class.

To instantiate an object

Remember that a class is like a template or blueprint which is used to create objects. An object is an **instance** of the class – so creating an object is called **instantiating**.

Important: You instantiate the object in your main file (ie not the file containing your class).

In the snake.py file ...

- Make a copy of your **`snake10.py`** file and name it **`snake11.py`**.
- Import the class you initialised – as shown below.

```
from food import Food
```

Then instantiate the object by calling the class.

```
apple = Food()
```

Food object named apple is instantiated.

Once you have your class(es) and object(s) set up, you can add the code that will be used to get the object to do things, look a certain way, etc.

For example, look at the code below.

```
apple.food_x, apple.food_y = apple.position_food(SCREEN_WIDTH,
SCREEN_HEIGHT, SNAKE_BLOCK)
```

This calls the `position_food` function in the `food.py` file. (Reminder: `apple` is the object based on the `Food` class). So every time you see `apple.`, this is referring to the object that was created based on the `Food` class (in the `food.py` file).

Continues next page

In the food.py file …

Below is what the function looks like in the food.py file (inside the Food class).

```
def position_food(self,SCREEN_WIDTH,SCREEN_HEIGHT,SNAKE_BLOCK):
        """ Sets random x and y coordinates for the food.
        The maximum range values are the screen dimensions
        minus the size of the food (so it doesn't ever go off
        screen). SNAKE_BLOCK is same size as food dimension.
        """
        food_x = round(random.randrange(SNAKE_BLOCK,
        SCREEN_WIDTH - SNAKE_BLOCK)/MOVE_BLOCK)*MOVE_BLOCK
        food_y = round(random.randrange(SNAKE_BLOCK,
        SCREEN_HEIGHT - SNAKE_BLOCK)/MOVE_BLOCK)*MOVE_BLOCK
        return food_x, food_y
```

- Add the rest of the code pertaining to the apple in the food.py file (eg the image used for the apple)
- Update your main file to call the relevant function(s). If you get stuck, refer to the sample answers **snake11_EXCEL.py** and **food.py** (in the **Workbook Sample Answers** folder – refer to the **Snake – 11** subfolder). Look for the comments marked '**Added for OOP**' to see all the code added to implement the game using an object-oriented programming approach.

  Note: If you do not have access to the **Workbook Sample Answers** folder, ask your teacher for this file.

More on Object-Oriented Programming (OOP)

For more on object-oriented programming, refer to the following websites:

- https://realpython.com/python3-object-oriented-programming/
- https://www.programiz.com/python-programming/object-oriented-programming
- https://www.tutorialspoint.com/python/python_classes_objects.htm
- https://www.datacamp.com/community/tutorials/python-oop-tutorial
- https://pythonprogramming.net/object-oriented-programming-introduction-intermediate-python-tutorial/

or do an internet search using keywords: **Python object oriented programming**