technocamps

Python Best Practice & Guidance

for Computer Science GCSE

Practitioner Workbook

Llywodraeth Cymru Welsh Government
Prifysgol Abertawe Swansea University
Cardiff University Prifysgol Caerdydd
Prifysgol Bangor University
Cardiff Metropolitan University | Prifysgol Metropolitan Caerdydd
University of South Wales Prifysgol De Cymru
Medr Y Comisiwn Addysg Drydyddol ac Ymchwil Commission for Tertiary Education and Research
institute of CODING in wales technocamps
Prifysgol Aberystwyth University
Prifysgol Wrecsam Wrexham University
Prifysgol Cymru Y Drindod Dewi Sant University of Wales Trinity Saint David

Introduction

technocamps

Welcome to the Python Practitioner Workbook! This book is intended to serve as a useful companion to teaching the Computer Science GCSE. In this book you will find 3 programs; a relatively simple Hangman game to help engage the learners in learning to code, a simple shop program that teaches all the fundamentals, and an advanced Tkinter shop program which is broken and must be fixed.

By using these 3 programs and more, we hope that all learners can feel sufficiently confident in Python to successfully fix a program during their examination! Additionally, there are pages dedicated to best practices and conventions within Python - while these may not be strictly required for GCSE, by adhering to them you will prepare your learners for a future in tech.

Understanding the Code

- **`Bold code`** is new code to be added/amended.
- *`Italic code`* is code that can/should be wrote on a single line (code CAN split over multiple lines so long as the line ends with a "connector" i.e. `(` or `,` or `+` etc.)
- The tildes (~ ~ ~) denote where there is more code that has not been printed
- Code has been coloured for readability and has more colours than IDLE.
- The correct indentation of the code is a necessity.

Online Resources

You can find all of our online resources and materials on our website! Scan the QR code or visit **tc1.me/educonf2025**

Get in touch!

We would be happy to discuss any questions or queries of the content mentioned within this workbook. Email us info@technocamps.com

Content

technocamps

Python Good Practice (PEP8 Guide)

Naming Conventions

Variables: `my_well_named_variable`

Constants: `MY_WELL_NAMED_CONSTANT`

Functions: `my_well_named_function`

Modules and Packages: `mywellnamedmodule` or `my_well_named_module`

Classes: `MyWellNamedClass`

Modules and Packages may use underscores if it improves readability - but ideally should be exclusively lowercase.

The Look

No Unnecessary Whitespace:

`add(x, y)` not `add ( x , y )`

Tabs Not Spaces!

This is actually the opposite of convention, which says indentation should always be exactly 4 spaces, but we find that is harder for learners to grasp!

Line Length should be no more than 80 characters!

This is to make your code readable on small screens, in windowed view, on vertical screens, when formatted in browsers etc...

No Commas

```
import math
import random
```

not

```
import math, random
```

```
name = "dan"
age = 30
```

not

```
name, age = "dan", 30
```

Python can do this but it is generally seen as bad practice! There are specific circumstances where it is accepted.

No Magic Numbers

```
SENSIBLE_NAME = 30
if my_variable > SENSIBLE_NAME:
```

not

```
if my_variable > 30:
```

0 and 1 are allowed as magic numbers!

No Errors

Possible errors should always be handled!

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
```

Commenting

Good use of commenting is essential!

This should be within reason - not every line of code requires a comment, the code itself should be self-evident. However, complicated iterations/selections may require clarification.

```
#This iterates through list and does something
```

Use DocStrings!

At the top of every module, function, and class, describe the purpose of that module, function or class for the user! This should not explain the code, just the general idea, what parameters must be provided, and what it returns!

```
"""
This function does this thing.
Args: n (int): It does it to this integer n
Returns: new_n (int): It returns the thing
"""
```

Small Scope

Functions and Modules should have a small restricted scope!

It is better to have lots of small functions that each perform simple tasks and can be reused. This also goes for the modules in your Python package.

This helps you adhere to the Python Community Standards:

DRY (Don't Repeat Yourself), Simple Is Better Than Complex, Readability Counts.

Python File Structure

```
"""
module docstring
"""

import standard_library
import thirdparty_library
import project_modules


MODULE_LEVEL_CONSTANT = 0


def all_functions(parameters):

    """
    function docstring
    """

    FUNCTION_LEVEL_CONSTANT = 0

    function_level_variable = 0


def allClasses:

    """
    class docstring
    """


main()
```

This is only the logical and necessary order of elements in a Python file.

Not all elements have to be incorporated in every file - and as mentioned small reusable modules are better!

Module Level Variables (Global) are possible but generally bad practice.

main() should always call the main function that begins the program.

In other words there should be no code floating around at the bottom of the python file - the code that starts the program.

Python Project Structure

```
my_package/
├── my_package/              # Main package directory
│   ├── module1.py           # Module 1
│   ├── module2.py           # Module 2
│   ├── subpackage/          # Subpackage directory
│   │   ├── submodule1.py
│   │   └── submodule2.py
├── tests/                   # Tests directory
│   ├── test_module1.py
│   └── test_module2.py
├── resources/               # Resources directory
│   ├── input_data.txt
│   └── output_data.csv
├── README.txt               # Documentation
└── requirements.txt         # List of dependencies
```

This is only a basic overview of the file structure you may encounter in Python projects - whether they are libraries you have downloaded or online representations such as on GitHub.

There are many more files and directories that may be included.

Note that creating a Package of this complexity, or recalling this structure, is not a requirement of the GCSE CS course.

This is simply a useful overview.

PEP 8 Style Guide:
peps.python.org/pep-0008/

Hangman

technocamps

Background

This task is intended as a fun project for learners to learn Python and grow in confidence using it.

The project demonstrated here will only output to the Shell, but there is no reason that this couldn't be recreated in Tkinter (with relative ease), or that it couldn't use the Turtle library to draw the graphics! The code for a Tkinter version has been included for reference.

Materials

The materials for this task - which includes both the completed program for reference by the educator and a wordlist for the learners, can be found at the following link: **tc1.me/educonf2025**

Notes

Task 1 - Random Word

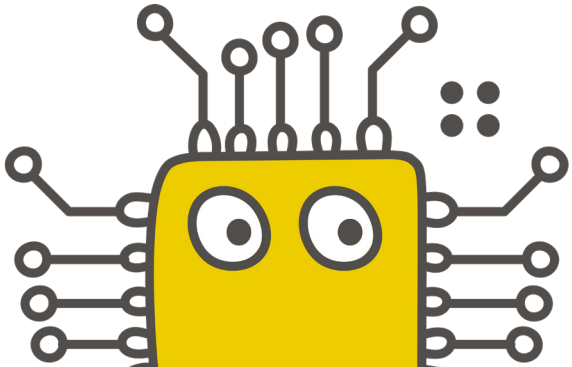
Task Description

To begin, learners must create a new Python file (i.e. **'hangman.py'** , this will be the only module in this activity - however a good bonus task would be to split the file over multiple modules!).

The first task is to generate a random word, that is after all the key to hangman.
A word list is provided so that learners don't have to write an exhaustive list.

Task 1 Code

```
import os
from random import randint

def random_word():

    word_file = open(os.path.dirname(
                          os.path.abspath(__file__)) +
                          "/wordlist.txt","r")
    word_data = word_file.read()
    word_list = word_data.split("\n")

    number = randint(0, len(word_list))
    word = word_list[number]

    return word
```



Task 2 - Game Start

Task Description

Now that we know we have the basic element of Hangman, we can start thinking through the logic of implementing it.

We need to start the game, but in order to show any representation to the screen (**i.e. _ _ _**) we will need to have generated the word. But maybe we also want to be able to replay the game! So we will make two functions, one called `main()` and the other called `game_start()`. At the start of `game_start()` we will declare the empty list of guesses, the random word and the representation - these should also happen when we replay the game.

Task 2 - Code

```
def main():

    print("Welcome to Hangman!")

    game_start()

def game_start():

    guessed = []

    word = random_word()

    progress = "_ " * len(word)
```

Task 3 - Print Out

Task Description

While we want to start implementing the gameplay logic itself, we also need to show each step to the player, or it will be unclear that anything has happened. We need to print the current game **state** at every step. Also, we should print this before starting the gameplay.

Task 3 - Code

```
def game_start():

    ~ ~ ~
    print_out(guessed, progress)


def print_out(guessed, progress):

    print("\n" + progress)

    guesses_str = ""

    for i in guessed:

        guesses_str = guesses_str + i + " "

    print("\nGuessed Letters:\n" + guesses_str)
```

Task 4 - Gameplay Logic

Task Description

Now comes the gameplay logic. We need the user to guess a letter, then:

`if` that letter is in the word we should replace the `"_ "` for all instances of the letter in `progress` with the letter itself

We will implement more complex requirements once the basic logic works

Task 4 Code

```
def check_word(word, progress, guess):

    for letter in word:

        if letter == guess:

            index = word.index(letter)

            i = index * 2

            progress = progress[:i] + word[index]
                       + progress[i+1:]

    return progress
```

Note that we must multiply the `index` by 2 for `progress` as we print 2 characters per letter (`"_ "`).

Task 5 - Gameplay Loop

Task Description

We must have a gameplay loop so that the user is able to guess all letters of the word. Each letter entered by the user must be: checked, added to the list of guessed letters, and printed, until all letters in the word have been found.

Task 5 - Code

```
def guessing(word, progress, guessed):

    while "_" in progress:

        guess = input("\nEnter a letter: ")

        progress = check_word(word, progress, guess)

        guessed.append(guess)

        print_out(guessed, progress)
```

Check Gameplay

Check the program to see if **a)** it works and **b)** whether it has any issues.

Task 6 - Errors

Task Description

By testing your game you will notice a few errors, such as:

- Users can enter the same letter more than once
- Users can enter non-alphabetical characters, multiple characters...
- Users have infinite "lives"

We will solve each of these in turn.

Task 6 - No Repeating Letters

```
def check_word(word, progress, guess):
    for letter in word:
        if letter == guess:
            ~ ~ ~
            word = word[:index] + " " + word[index+1:]
    return progress
```

Note that by adding this line in `check_word`, we gradually replace all correctly guessed letters in the `word` with spaces, so that they'll never again match the `guess`. That doesn't stop the letter from being guessed or being added to `guesses`, but we will implement that with the next step.

Task 6 - Validation

```
def guessing(word, progress, guessed):

    global lives

    while ("_" in progress) and (lives > 0):

        guess_check = True
        while guess_check:

            guess = input("\nEnter a letter: ")

            if len(guess) != 1:
                print("Enter only 1 letter!")

            elif guess.isalpha() == False:
                print("Enter a letter!")

            elif guess in guessed:
                print("You've tried that letter!")

            else:
                guessCheck = False

        progress = check_word(word,progress,guess)

        guessed.append(guess)

        print_out(guessed, progress)
```

Task 6 - Lives

Note we added the check for lives, so that the gameloop only runs while the user has lives remaining, but we still must implement the logic.

```
def game_start():

    global lives
    lives = 9

    ~ ~ ~

def check_word(word, guess, progress):

    global lives
    letter_found = False

    for letter in word:
        if letter == guess:

            letter_found = True

            ~ ~ ~

    if letter_found == False:
        lives -= 1

    return progress
```

Task 7 - Extensions

Task Description

Plenty of potential extensions exist for this game, some have been intentionally excluded from this code as they are very simple, others are more complex or build on foundational knowledge.

Intentionally Excluded (Simple):

- Add success and failiure messages for when the user guesses the word / runs out of lives!
- Put the entire game on a loop so that the user may choose to play again without having to relaunch the program.

Standard:

- Add 2 player functionality and/or a scoreboard so that players can track their wins.
- Add a difficulty option (chooses words based on length?)

Advanced:

- Use the Turtle (or other graphical) library to draw the hangman as the user loses each life!
- Use the Tkinter library to give a GUI to the entrie program!

Notes

Tkinter & Canvas

```
import os
from random import randint
import tkinter as tk
from tkinter import messagebox

def main():
    global root, canvas

    # Create the main window
    root = tk.Tk()
    root.title("Hangman")

    #Initialise game
    game_start()
    root.mainloop()


def game_start():
    global lives, guessed, word

    # Define lives, empty list of guessed letters,
    # and word to guess
    lives = 6
    guessed = []
    word = random_word()

    gui_start()
```

Tkinter & Canvas

```
def random_word():

    word_file = open(os.path.dirname(
                    os.path.abspath(__file__)) +
                    "/wordlist.txt","r")
    word_data = word_file.read()
    word_list = word_data.split("\n")

    number = randint(0, len(word_list))
    word = word_list[number]
    return word

def gui_start():

    global root, lives, guessed, word
    global canvas, word_label, lives_label
    global guessed_label, guess_entry

    # Create the canvas for drawing the hangman
    canvas = tk.Canvas(root, width=300, height=300)
    canvas.grid(column=0, row=0)

    # Create Retry and Exit Buttons
    retry_button = tk.Button(root, text="Retry",
                             command=lambda:retry_game())
    retry_button.grid(column=0, row=5)

    exit_button = tk.Button(root, text="Exit",
                             command=lambda:exit_game())
    exit_button.grid(column=0, row=6)
```

Tkinter & Canvas

```
# Allows user to press enter-key to input
root.bind("<Return>", lambda event:check_guess())

# Draw the scaffold
canvas.create_line(20, 280, 120, 280)
canvas.create_line(70, 280, 70, 20)
canvas.create_line(70, 20, 170, 20)
canvas.create_line(170, 20, 170, 50)

# Create label for displaying the word
word_label = tk.Label(root,
    text=" ".join(["_" for letter in word]))
word_label.grid(column=0, row=1)

# Create label for displaying lives remaining
lives_label = tk.Label(root,
    text="Lives remaining: {}".format(lives))
lives_label.grid(column=0, row=2)

# Create label for displaying letters guessed
guessed_label = tk.Label(root,
    text="Guessed letters: ")
guessed_label.grid(column=0, row=3)

# Create entry for the user to guess a letter
guess_entry = tk.Entry(root)
guess_entry.grid(column=0, row=4)
```

Tkinter & Canvas

```
def check_guess():

    global root, lives, guessed, word, word_label
    global lives_label, guessed_label, guess_entry

    guess = guess_entry.get().lower()
    guess_entry.delete(0, tk.END)

    # Check if the guess is a single letter
    # or if the guess has already been guessed
    if (len(guess) != 1) or (not guess.isalpha()) or
    (guess in guessed):
        return

    # Add guess to guessed
    guessed.append(guess)
    guessed_label.config(
                text="Guessed letters: {}".format(
                        " ".join(guessed)))

    # Check if the guess is in the word
    if guess in word:
        letter_list = list(word_label["text"])

        for i in range(len(word)):
            if word[i] == guess:
                letter_list[2*i] = guess

        word_label.config(text="".join(letter_list))
```

Tkinter & Canvas

```
# Check if the user has won
    if "_" not in word_list:
        messagebox.showinfo("Hangman", "You win!")
        guess_entry.config(state=tk.DISABLED)
        return

# If guess not in word, decrement lives
else:
    lives -= 1
    lives_label.config(
      text="Lives remaining: {}".format(lives))

    # Draw the hangman
    if lives == 5:
        canvas.create_oval(140, 50, 200, 110)
    elif lives == 4:
        canvas.create_line(170, 110, 170, 170)
    elif lives == 3:
        canvas.create_line(170, 130, 140, 140)
    elif lives == 2:
        canvas.create_line(170, 130, 200, 140)
    elif lives == 1:
        canvas.create_line(170, 170, 140, 190)
    elif lives == 0:
        canvas.create_line(170, 170, 200, 190)
        messagebox.showinfo("Hangman",
    "You lose! The word was '{}'".format(word))
        guess_entry.config(state=tk.DISABLED)
```

Tkinter & Canvas

```
def retry_game():

    global canvas
    global guessed_label, word_label, lives_label

    # Destroy canvas and clear labels
    canvas.destroy()
    guessed_label.config(text="")
    word_label.config(text="")
    lives_label.config(text="")

    #Restart Game
    game_start()


def exit_game():

    global root

    #Destroy root (i.e. entire GUI)
    root.destroy()


main()
```

Note that this program uses an excessive number of global variables, this is bad practice and should be avoided! However, it does make Tkinter far simpler (and simpler to follow). All variables in this program should ideally be passed as parameters instead.

PyShop

technocamps

Background

This task was previously intended to demonstrate all requirements of the **Computer Science GCSE** coursework, however this no longer exists.

The task is still worthwhile in understanding how to build a large Python program, and it covers many core concepts. Additionally, the **Sample Materials** for the new **CS GCSE** uses code similar to that of the previous coursework, and it may be reasonable to assume that they will continue to use this style of system for future assessments.

Materials

The materials for this task - which includes both the completed program for reference by the educator and broken program for learners to fix, as well as a presentation to aid with content delivery - can be found at the following link: **tc1.me/educonf2025**

Notes

Task 1 - Login

Task Description

To begin, learners must create a new Python file (i.e. **'pyshop.py'** , this will be the only module in this activity - however a good bonus task would be to split the file over multiple modules!).

The first task is to create a simple login system using the terminal (we will not make use of Tkinter in this task). The login system must store both the username and password (as variables for now), accept user input for the username and password and check whether they match, and finally print a login success or failure message for the user.

Task 1 Code

```
stored_username = "admin"
stored_password = "pass"

in_username = input("Enter username: ")
in_password = input("Enter password: ")

if (stored_username == in_username) and
(stored_password == in_password):
    print("Welcome!")
else:
    print("Login failed!")
```

Task 2 - Menu

Task Description

The next step is to implement a main **menu** for the user once they have logged in. This will allow users who have logged in successfully make use of the shop system - it should therefore have a few features.

For now we will not implement these features, we will simply **stub** them - this is when you leave room for a planned future feature in your code, while still allowing that code to be run (it can either do nothing or notify you that it has run). It is what we did previously with the print statements stubbing the login, it is an incredibly useful tool for programmers!

Notes

Task 2 - Code

```
~ ~ ~

if (stored_username == in_username) and
(stored_password == in_password):

    print("Welcome!")

    menu_choice = input("""
    Please select an option:
    1. View
    2. Add
    3. Delete
    """)

    try:
        selection = int(menu_choice)
        if selection == 1:
            print("Menu -> View")
        elif selection == 2:
            print("Menu -> Add")
        elif selection == 3:
            print("Menu -> Delete")
        else:
            print("Invalid selection!")
    except ValueError:
        print("Must enter a valid integer!")

else:
    print("Login failed!")
```

Task 3 - Functions

Task Description

Now that the menu is working, we'll need to begin replacing the stubs with implementations. However, this is going to result in one very large, hard to follow, messy program!

To avoid this, we can begin to separate our code into multiple functions. Ideally, each function will have only one purpose and be named appropriately; this will help keep our code readable and easy to reuse (if necessary).

Let's begin by splitting the login and menu into two separate functions! Also, to keep to Python conventions, we will begin the program by calling a function called main().

Note: we would also ideally separate our functions across multiple self-contained modules, as to avoid having one very large Python script!

Notes

Task 3 Code - Functions (Menu)

For the `menu()` function, we will take the menu implementation from inside `login()`, and place it within its own definition.

```
def menu():

    menu_choice = input("""
    Please select an option:
    1. View
    2. Add
    3. Delete
    """)

    try:
        selection = int(menu_choice)
        if selection == 1:
            print("Menu -> View")
        elif selection == 2:
            print("Menu -> Add")
        elif selection == 3:
            print("Menu -> Delete")
        else:
            print("Invalid selection!")

    except ValueError:
        print("Must enter a valid integer!")
```

Task 3 Code - Functions (Login)

For the login function, we will place the function within a definition, removing the menu that we previously added. We will not call the menu function yet, but add two return statements, which we will see the purpose of when we implement the `main()` function.

```
def login():

    stored_username = "admin"
    stored_password = "pass"

    in_username = input("Enter username: ")
    in_password = input("Enter password: ")

    if (stored_username == in_username) and
    (stored_password == in_password):
        print("Welcome!")
        return True

    else:
        print("Login failed!")
        return False
```

Task 3 - Functions (Main)

We will also add a main function at the top of our script (this is convention), and then we must call it from the bottom of our script (i.e. the script is sandwiched between). Now when Python reaches the bottom it calls `main()` which in turn calls `login()` and `menu()`.

```
def main():
    login()
    menu()


~ ~ ~


main()
```

Notes

Task 4 - Loops

Task Description

Now the code looks a lot tidier, and it will be significantly easier to both modify and add to it. However, before adding features there's still a new issue - whether the user enters the username or password correctly they advance to the menu. And after selecting an item in the menu the program ends.

Both of these issues can be solved with loops!

Task 4 - Loops (Main)

The first while loop will keep the user stuck on login until a successful login occurs, while the second while loop keeps the user stuck in the menu forever.

```
def main():

    login_successful = False

    while not login_successful:

        login_successful = login()

    in_menu = True

    while in_menu:

        menu()
```

Task 5 - Items

To implement features into this menu, a data structure to hold items for sale must be constructed. There are many ways to do this, but this example uses a dictionary to store the details of each item, with all items nested within a list. This is declared **global** so that it can be accessed anywhere.

This data **must** be loaded in from an **external file** (and saved there whenever any changes are made). It cannot be hard-coded as this would make any updates temporary (i.e. when the program is closed, upon relaunch it will load the same hard-coded list).

If it had been hard-coded, the structure would look like this:

```
items = [     {
                        "Name": "Coca-Cola",
                        "Price": 1.50,
                        "InStock": 50
              },{
                        "Name": "Irn-Bru",
                        "Price": 1.20,
                        "InStock": 30
              },
                        ...
              }
          ]
```

Task 5 - Items (Load)

```
def load_items():

    global items

    stock_file = open('stock.csv','r')

    items = []

    for line in stock_file:
    item = []

    for column in line.strip().split(','):
            item.append(column)

        items.append(     {
                          "Name":     item[0],
                          "Price":    int(item[1]),
                          "InStock":  int(item[2])
                          })

    stock_file.close()
```

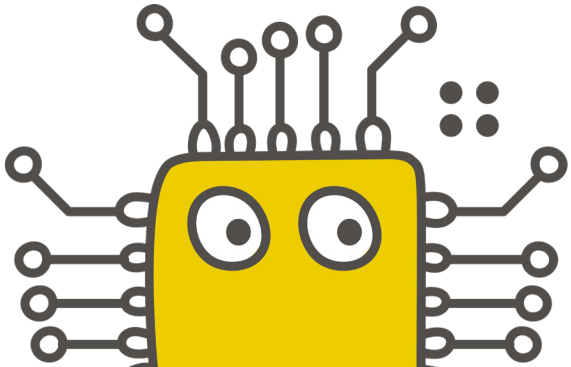
Task 5 - Items (Save)

```
def save_items():

    global items

    stock_file = open('stock.csv','w')

    for item in items:

        line = item["Name"] + "," +
               str(item["Price"]) + "," +
               str(item["InStock"]) + "\n"

        stock_file.write(line)

    stock_file.close()
```

Task 5 - Items (csv)

The learners will have to continue or make a new **csv** file, and fill it with items in the following format:

```
Coca-Cola,1.5,50
Irn-Bru,1.2,30
...
```



Task 6 - View

Task Description

Now we can start implementing functions into the menu. We'll start with a simple function to view all the items.

Before this will work, we will have to call the `load_items()` function in `main()`.

Task 6 - View

```
def view_all():

    global items

    for item in items:

        print("Name: " + item["Name"])
        print("Price: " + str(item["Price"]))
        print("Stock: " + str(item["InStock"]) + "\n")
```

Task 6 - View (Menu)

```
def menu():

    ~ ~ ~

    try:
        selection = int(menu_choice)
        if selection == 1:

            view_all()

    ~ ~ ~
```

Task 6 - View (Main)

```
def main():

    login_successful = False
    while not login_successful:
        login_successful = login()

    load_items()

    in_menu = True
    while in_menu:
        menu()
```

Task 7 - Add

Task Description

Now we'll implement the add function to allow new items to be added to stock. This will require us to write the items to file after making changes. It is also essential to call this function from the menu!

Note: this should also include error handling in case the user input is invalid, this has been left out intentionally.

Task 7 - Add

```
def add_item():

    global items

    new_item = dict()

    new_item["Name"] = input("Item name: ")
    new_item["Price"] = float(input("Item price: "))
    new_item["InStock"] =
                    int(input("Number in stock: "))

    items.append(new_item)

    save_items()
```

Task 8 - Delete

Task Description

Now we'll implement the delete function to allow items to be removed from stock. This will require us to rewrite the items file after making changes. It is also essential to call this function from the menu!

Task 8 - Delete

```
def delete_item():

    global items
    item_name = input("Name of item to delete: ")

    for item in items:
        if item["Name"] == item_name:

            print("Name: " + item["Name"])
            print("Price: " + str(item["Price"]))
            print("Stock: " + str(item["InStock"]) +
                  "\n")

        confirm = input("Delete item? Y/N: ")
        if confirm.upper() == "Y":
            items.remove(item)
            save_items()
```

Task 9 - Extensions

Task Description

There are countless extensions that can be done to this program, and we would encourage learners to attempt their own extensions to encourage self guided learning. Practice, trial and error are the best way to learn a programming language.

Possible extensions include:

- An update function to change the name, price or quantity of an item.
- A method to purchase items from the shop - this should ideally reduce the quantity and keep a running total of income.
- Multiple methods of searching the items (for use in delete and update)

More complex extensions include:

- Printing/saving a transaction log upon logging-out, which must keep track of all sales made during the session. Even better if multiple different items can be purchased per transaction!
- Assuming multiple different items per transaction, print a customer receipt.
- Implementing the entire system using Tkinter.

Notes

PyShop Cheat Sheet

technocamps

Python Syntax

Action	Python Code
Print something	`print("This will be printed")`
Assign a variable	`my_variable = 42` `my_other_variable = "Hello"`
if/else	`if age > 17:` `    print("You are an adult")` `else:` `    print("You are not an adult")`
Getting input	`name = input("What is your name?")` `age = int(input("What is your age?"))`
Cast input	`user_input = input("age:")` `age = int(user_input)` `print(age)`
Handling exceptions	`user_input = input("age:")` `try:` `    age = int(user_input)` `    print(age)` `except ValueError:` `   print("you did not enter an int")`
Functions	`def print_welcome(name):` `    print("Welcome", name)` `print_welcome("Casey")`

Python Syntax

Action	Python Code
Lists	`empty_list = []  /  empty_list = list()` `text_list = ["a", "b"]` `number_list = [1, 2, 3]` `mixed_list = ["a", 1, "b", 2]` `print(text_list[0])` `print(len(text_list))` `text_list.append("d")` `text_list[0] = "z"` `text_list.remove("b")`
2D Lists	`empty_list = [[],[]]` `my_list = [ [0,1,2,3,4] , [3,6,1] ]` `print(my_list[0][1])`
Dictionaries	`empty_dict = {}  /  empty_dict = dict()` `cars = {` `"Audi Q5" : 20000,` `"Volkswagen Polo" : 18000` `}` `print(list(cars.keys()))` `print(cars)` `print(cars["Audi Q5"])` `print(len(cars))` `cars["Ford KA"]=500` `cars["Audi Q5"]=10000` `cars.pop("Audi Q5")`

Python Syntax

Action	Python Code
while loop	`password = ""` `while password != "secret":` `    print("What is the password?")` `    password = input()` `print("You guessed it!")`
for loop (list)	`fruits = ["banana", "apple"]` `for f in fruits:` `    print(f)`
for loop (count)	`for i in range(10):` `    print("i   = ", i)` `    print("i*i = ", i*i)`
Reading a file	`f = open("Test1.txt", "r")` `for line in f:` `    print(line)` `f.close()`
Writing to a file	`f = open("Test2.txt", "w")` `f.write("This is written.")` `f.close()`
Append to a file	`f = open("Test3.txt", "a")` `f.write("This is appended")` `f.close()`
Create a file	`f = open("Test4.txt", "x")`

Fixing Tkinter

technocamps

Background

This task is intended as practice for the **Computer Science GCSE** curriculum.

This is a new version of the PyShop activity written using Tkinter. The aim of the task is to understand how the the program functions and fix all errors present within, while also improving the learners' ability to provide good commenting and documentation.

In this program we have tried to include all manner of programming techniques that have appeared in the **Sample Materials**. While not all of these techniques include errors that must be fixed, it is hoped that having to grasp the intended function of the code will aid learners in understanding the variety of Python programming styles.

Materials

The materials for this task - which includes both the completed program for reference by the educator and broken program for learners to fix, as well as a presentation to aid with content delivery - can be found at the following link: **tc1.me/educonf2025**

Task 1 - Fixing Login

Task Description

In the first task, learners must look at the **'gui_shop_login.py'** file and fix the **6 errors** present within. This is intended as an introductory task, as not to overwhelm by providing them with 4 broken modules of a large program.

However, to run and begin testing this file, learners must write a **main module** to call it. This is essential Python knowledge, as the majority of programs will be **decoupled** into modules; where a main module calls others and runs the program!

Main Module

Create a new Python file called **'gui_shop_main.py'**, saved in the same directory as the other modules, and include the following code:

```
from gui_shop_login import create_login

create_login()
```

Login Error #1

The stored username and password variables were left blank, both should have values to ensure only an authenticated user can access the program:

```
stored_username = "A.Username"
stored_password = "goodPassw0rd"
```

Login Error #2

The login check ensured that **either** the username **OR** password matched the stored variables. A login system should ensure that **both** match:

```
if (username_entry.get() == stored_username) and
(password_entry.get()== stored_password):
```

Login Error #3

The window title was nonsense, a well-designed program should have informative names for each window:

```
login_root.title("Shop Program")
```

Login Error #4

Any Tkinter elements must be placed within a Frame using a **geometry manager**, in this case, we're using **pack**.

```
title_label = Label(login_frame,
                    text = "Welcome to Shop Login!")
title_label.pack()
```

Login Error #5

In the GUI the password field appeared to the left of the password label. This seems like a mistake and should be rectified so that the label is on the left.

```
password_label = Label(password_frame,text="password")
password_label.pack(side = LEFT)
password_entry = Entry(password_frame)
password_entry.pack(side = RIGHT)
```

Login Error #6

When the **Submit Button** is pressed, it should run the `process_login` function (that means the `pointless_function` can be deleted).

```
submit_button = Button(login_frame,
                       text = "submit",
                       bg = "White", fg = "Red",
                       command = process_login)
```

Note that this function is provided without brackets (i.e. it is not calling the function), meaning it is only a **reference** to the function (i.e. a variable containing the function call). This is why process_login is **nested** within create_login, so that it is within scope to make a reference.

Try replacing the `command = pointless_function` with the `command = print("This function is pointless.")`, and see what happens (it is immediately called upon running the program and the button no longer does anything)!

Task 2 - Commenting

Task Description

The second task is to ensure that the code is well commented. The learners must add comments where needed in the **'gui_shop_login'** module - though this task should continue throughout.

Commenting is an essential programming skill, and is included as example questions in the new curriculum.

Best commenting practice:

- **Don't comment every line of code** - instead space-out sections of code and comment those chunks that do a particular thing.

- **Comments shouldn't explain what the code is doing** - that's what the code is for, they should only explain the purpose. This can be the tricky one to understand, so this example might help:

  - **Bad Comment**

```
# Prints out 'Login Successful' then on the next line 'Hi' along
# with the username entered by the user
print("Login Successful!\nHi ", username)
```

  - **Good Comment**

```
# Welcome message for user
print("Login Successful!\nHi ", username)
```

- **Functions should be commented** with """docs""" - these should explain the functions purpose and outline any arguments (written as **Args:**) and/or **Return:** statements present in the function.

Task 3 - Main Module Overhaul

Task Description

We need a way to **switch** between the login window and the shop face via the log-in/log-out buttons.

This is quite tricky to do when we're using Python **procedurally**, as we can run into errors by 'infinitely' switching windows (this is a RecursionError, as recursive functions are only permitted to go so many layers deep - Tkinter is **intended** to be used in an Object Oriented structure which removes many of the issues we come across!)

So, we'll need a **creative solution** to avoid the error! We need to **destroy** the login window **before switching** to the shop (so that no recursion occurs) - but if we do this within the login window the shop will never be called (as the window has been destroyed before calling the login)!

To get around this, we'll have to implement everything within 'gui_shop_main', by defining functions that destroy the current window and call the new window. This will require **passing** references to these functions into each window, as we previously saw in Login Error #6. Additionally, we will have to modify 'gui_shop_login' to accept and make use of this reference.

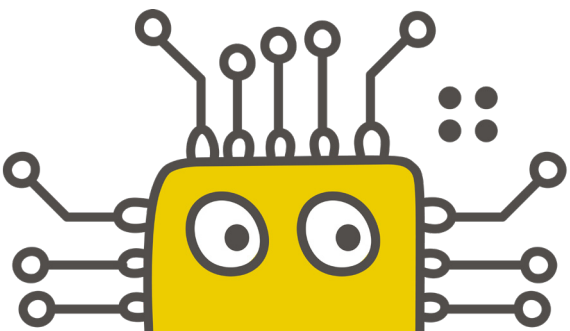
Notes

Main Module Overhaul

```
from gui_shop_login import create_login
from gui_shop_shopface import create_shopface

def switch_to_shop(login_root):
    login_root.destroy()
    create_shopface(switch_to_login)

def switch_to_login(shopface_root):
    shopface_root.destroy()
    create_login(switch_to_shop)

create_login(switch_to_shop)
```

Update Login Module

```
def create_login(switch_to_shop):

    ~ ~ ~

    if (username_entry.get() == stored_username) and
    (password_entry.get() == stored_password):

        switch_to_shop(login_root)
```



Task 2 - Commenting

Task Description

Now that the login window works as intended, learners can run the **'gui_shop_main.py'** file to test the program (they must have all files, including the **'gui_shop_shopface.py'**, **'gui_shop_filehandling.py'** and **'items.csv'** in the same directory).

This should now allow the user to log in and access the shop face, though this window will appear very broken. We will now have to fix the issues present in both **'gui_shop_shopface.py'** and **'gui_shop_filehandling.py'**.

This task can be provided as one large challenge, but learners may find it beneficial to solve the errors present one function at a time. This is a fairly large Python module, so it may feel overwhelming without guidance, as there are a total of **13 errors** to find and fix across 10 functions.

Notes

create_shopface - Shop Face Error #1, 2

Two errors exist within the create_shopface function (**'gui_shop_shopface.py'**).

The first of these requires the variables stock and total_sales to be declared as global. These variables are displayed within the GUI, so making them **global** and accessible by all functions is essential to having them update within the GUI.

The second error is that the stock variable is never set. This must be set by calling the load_stock function from **'gui_shop_filehandling.py'**.

```
def create_shopface(switch_to_login):

    # stock and total_sales defined as global...
    global stock
    global total_sales

    # shopface_frame is defined as global...
    global shopface_frame

    # Initialize stock dictionary by loading...
    stock = load_stock()
```

get_stock_file - FileHandling Error #1

To successfully load the stock into the shopface window, **2 errors** must be fixed within **'gui_shop_filehandling.py'.** The first of these simply requires the correct file name to be provided (the path is generated dynamically and should work on all devices).

```
def get_stock_file(mode):

    directory = os.path.dirname(
                                os.path.abspath(__file__))

    stock_path = os.path.join(directory, "items.csv")
```

get_stock_file - FileHandling Error #2

The second requires that the price is loaded from the csv **as a float** - this is because it is a decimal value and cannot be typecast to an integer.

```
def load_stock():

    ~ ~ ~

    price = float(item[2].strip())
```

create_menu_bar - ShopFace Error #3

The third error can be found within the create_menu_bar function (**'gui_shop_shopface.py'**).

As seen in the login (specifically the **submit_button**), a command must be given to the button. In this case the **switch_to_login** function passed from **'gui_shop_main.py'** is provided, so that this window is destroyed and the login window called upon logging out.

Note the use of the **lambda** function, this allows a function to be passed to the button without immediately being called - in the login we solved this with a reference to a nested function, but here we require a parameter to be passed, so a reference would not suffice. This is a better solution, as nested functions are against convention.

```
def create_menu_bar(switch_to_login, shopface_root):

    ~ ~ ~

    log_out_button = Button(
                     menu_frame,
                     text = "Log Out",
                     bg = "White", fg = "Red",
                     command = lambda:
                     switch_to_login(shopface_root))
```

create_stock_table - ShopFace Error #4, 5

There are **6 errors** to be found within the create_stock_table function (**'gui_shop_shopface.py'**), these appear consecutively and have been grouped into errors #4 & 5. All of these errors are simple mistakes in the value of which row to be displayed on.

```
def create_stock_table():

    ~ ~ ~

    id_label = Label(table_frame,
                     text = "Product ID").grid(
                     row = 0, column = 0)

    name_label = Label(table_frame,
                     text = "Name").grid(
                     row = 0, column = 1)

    ~ ~ ~

    for i, item in enumerate(stock):
        id_label = Label(table_frame,
                   text = item).grid(
                   row = i + 1, column = 0)

        name_label = Label(table_frame,
                   text = stock[item]["Name"]).grid(
                   row = i + 1, column = 1)
```

create_purchase_bar - ShopFace Error #6, 7

There are **2 errors** to be found within the `create_purchase_bar` function (**'gui_shop_shopface.py'**). The first of these is that the number of decimal places shown for the total_sales in the GUI is incorrect. This is defined with an f-string, so the formatting must be corrected.

The second error is again in the button. This button is even more complex than the last as the lambda function now contains an if statement - it checks whether the `process_purchase_attempt` function returns True, and if so it runs the `process_purchase function` (else it does None). This requires the correct parameter is passed to `process_purchase_attempt`.

```
def create_purchase_bar():

    ~ ~ ~

    sales_value = Label(purchase_frame,
            text = str(f"{total_sales.get():.2f}"))
    sales_value.grid(row = 1,column = 2)

    # Create a purchase button to make purchases
    purchase_button = Button(purchase_frame,
                text = "Make Purchase",
                bg = "White", fg = "Green",
                command = lambda: process_purchase(
                sales_value, purchase_entry.get())
                if process_purchase_attempt(
                purchase_entry.get()) else None)
```

update_stock_table - ShopFace Error #8

Only **1 error** exists within update_stock_table (**'gui_shop_shopface.py'**). After the current table is destroyed, another table must be created in its place.

```
def update_stock_table():

    global table_frame
    table_frame.destroy()
    create_stock_table()
```

process_purchase_attempt - ShopFace Error #9

Only **1 error** exists within the process_purchase_attempt function (**'gui_shop_shopface.py'**). If the attempt is unsuccessful due to an invalid input, then a **meaningful** error should be provided to the user..

```
def process_purchase_attempt(user_input):

    ~ ~ ~
    except (ValueError, KeyError):
        messagebox.showwarning(
                                "Key Error",
                                "Enter a Valid Key!")
        return False
```

process_purchase - ShopFace Error #10

Only **1 error** exists within process_purchase function (**'gui_shop_shopface.py'**). Whenever **global variables** are used within a Python function, they must be **redeclared** within that function (this is to inform Python that the global variable is being invoked, not a local variable with the same name).

```
def process_purchase(sales_value, user_input):

    #...
    global stock
    global total_sales
```

write_stock - FileHandling Error #3

The final error exists back in 'gui_shop_filehandling.py' within the write_stock function. In order to write any changes to file, the file must be opened in write mode, not read mode.

```
def write_stock(stock):

    stock_file = get_stock_file('w')
```

Notes

Tkinter Cheat Sheet

technocamps

Tkinter Syntax

Action	Python Code
Import	`import tkinter as tk` <- Better OR `from tkinter import *` <- Simpler While it is simpler to import all functions into local scope (you don't have to preface them with `'tkinter.'`) it does not import submodules (such as messagebox) and functions can be overwritten by having the same name. The rest of this cheat sheet will use the better import.
Define Main Window	`root = tk.Tk()` The main window to hold the Tkinter GUI must be created, we generally call this root. If we were to have multiple windows, we could use `tk.Toplevel()` to create new windows, but we only need one root for the main program.
Window Options	`root.title("Titlebar - Top of Window")` `root.geometry("640x480")` These will set the title and dimensions for the entire window. Many more options exist and many of the Frame options can be used directly on the Window too.
Define Frames	`my_main_frame = tk.Frame(root)` The Frame is the basic building block for content within the Window. We assign a geometry manager to the Frame to choose how we place the Tkinter widgets within. We can also place Frames within Frames!

Geometry Managers

Note that Geometry Managers are versatile in their usage. They are used to place widgets and/or Frames inside of other Frames or Windows.

It is not necessary to stick with a single geometry manager, and they can all be used within a program - however each Frame and Window must use a single manager (but a Frame placed within it could use another).

All geometry managers have more options and fine control than demonstrated!

Tkinter Syntax

Action	Python Code
Geometry Manager - **Pack**	`my_main_frame.pack()` `my_main_frame.pack(side = LEFT)` Pack is the simplest of the geometry managers and will simply pack widgets into the space provided with minimal effort (and with minimal fine tuning).
Geometry Manager - **Grid**	`my_main_frame.grid(row = 0,column = 0)` Grid is the nice middle-ground of the geometry managers and allows you to place widgets as if they are on a grid (this will size to your window, but you can have as many rows/columns as you like).
Geometry Manager - **Place**	`my_main_frame.place(x = 0, y = 0,` `width = 15,` `height = 10)` Place is the most complex geometry manager, but also allows for the most precise sizing and placement of widgets.

Tkinter Syntax

Action	Python Code
Destroy Frame	`my_frame.destroy()` This destroys the current Frame or Window. It is useful for options menus or login screens etc.
Main Loop	`root.mainloop()` This is essential as it defines the program loop (i.e. it is what keeps the entrie GUI within a loop so that text can be entered, buttons can be pressed, etc.). Generally only one mainloop need be defined at the bottom of the code that defines the main program/window.
Labels	`my_label = Label(my_frame,` `text = "Hello there")` `my_label.grid(row = 0, column = 0)` A label is simply a piece of text that appears on the GUI - here the label "Hello there" will be placed in the top left corner of `my_frame`.
Entry Fields	`my_entry = Entry(my_frame)` `my_entry.grid(row = 1, column = 1)` An entry field allows the user to enter text. Once the text has been entered, we could have a button the gets the contents of `my_entry`. Note that this entry field is currently in the bottom right corner - so long as we don't add any new widgets into row or column 2.

Tkinter Syntax

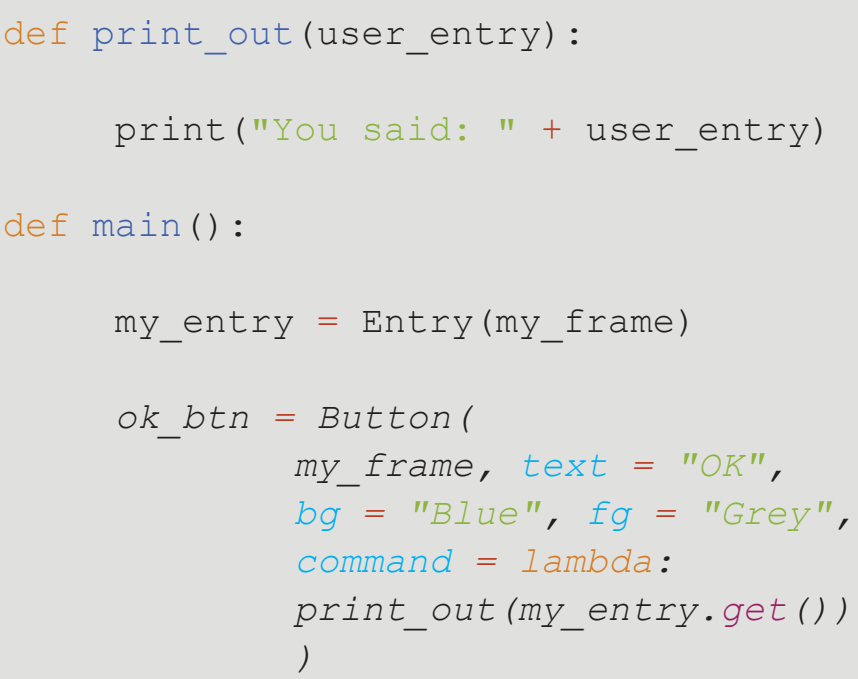
Action	Python Code
Button (**simple**)	``` def main(): def nested_ok(): print("This is a nested function that just prints - OK!") ok_btn = Button(my_frame, text = "OK", bg = "Blue", fg = "Grey", command = nested_ok) ```

To have a button do anything, it must be provided a command - however, providing it a function would cause that function to be immediately called (as opposed to called when the button is clicked).

So instead, we can pass a **reference** to a **nested function** (a **reference** is essentially a variable - we don't include the brackets - and the **function** must be **nested** inside the current function so that it is within scope!)

This works relatively well, but nested functions cannot be used elsewhere so they may result in repeat code. They can also harm readability if they are particularly long.

Tkinter Syntax

Action	Python Code
Button (**standard**)	

```
def print_out(user_entry):

     print("You said: " + user_entry)

def main():

     my_entry = Entry(my_frame)

     ok_btn = Button(
              my_frame, text = "OK",
              bg = "Blue", fg = "Grey",
              command = lambda:
              print_out(my_entry.get())
              )
```

By using a **lambda function**, we can avoid having to pass only a **reference** and can instead pass the entire function. The function will not be run immediately when prefaced with the **lambda** command.

This can be an essential requirement for any functions that take parameters! While clever workarounds to avoid passing parameters may be possible, it is far easier to just write `' lambda: '`.

Tkinter Syntax

Action	Python Code
Button (**advanced**)	 `def print_no(user_entry):` `  print("Number = " + user_entry)` `def is_no(user_entry):` `  if user_entry.isnumeric():` `  return True` `  else:` `  return False` `def main():` `  my_entry = Entry(my_frame)` `  ok_btn = Button(` `  my_frame, text = "OK",` `  bg = "Blue", fg = "Grey",` `  command = lambda:` `  print_no(my_entry.get())` `  if is_no(my_entry.get())` `  else print("Number only")` `  )` We can make our **lambda functions** much more complex by appending them with if statements! A lambda function is essentially a small, throwaway function that can be written on a single line - so they can be quite complex! This function will **only** print out what the user puts in the entry box **if** the content of the entry box is numerical, **else** it will specify that it only accepts numbers!

Tkinter Syntax

Action	Python Code
Message Box	`from tkinter import messagebox` `messagebox.showinfo("Title", "Message")` `messagebox.showerror("Title", "Message")` `messagebox.showwarning("Title",` `"Message")` `messagebox.showerror("Parse Error",` `"Numeric Only!",` `command = lambda:` `clear_entry())` Using the messagebox command in Tkinter allows us to use the default pop-up box of our system (i.e. on Windows will look like Windows pop-ups...). This is a great way to add professionalism to our project! The message box can just contain a simple title on the title bar and message in the box (the different functions info, error, warning etc. change the icon that appears!). However, we can get more complex by adding commands (via **lambda functions**) that run upon closing the box. There are countless variations to this, and many more types of boxes (`.askquestion, .askokcancel`) that allow us to return and process user input!