QA

Learn. To Change.

WELCOME

LIVE STAGE | Python in practice

v2.0.03.09.2024

Add some post its!

Press N to create your own sticky notes

Hello all!

Hi

Hi!

Hello World

Morning!

Hi

Draw something

Press P to create your art

Write something

Press T to express your thoughts

Hello

Hello again

Morning!

Name?

Where you work?

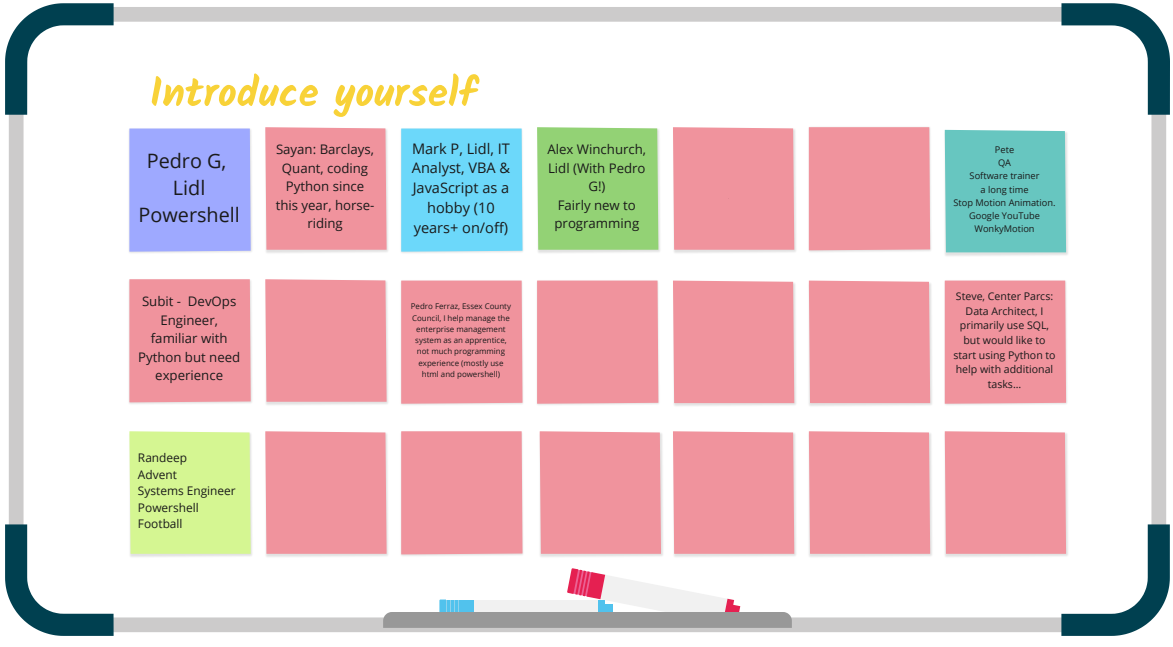
What you do?

Programming experience?

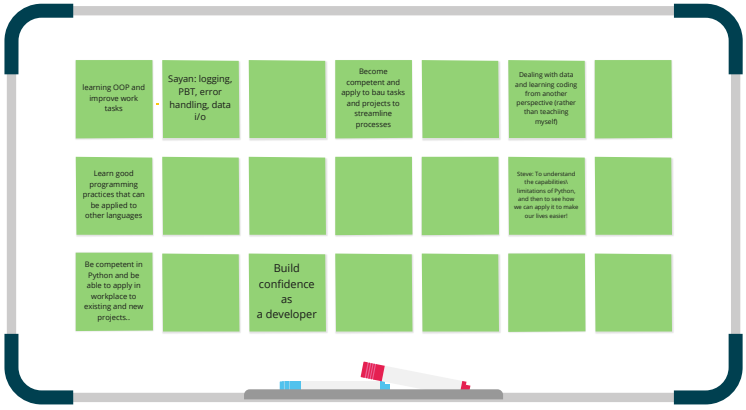
Sports?

Hobbies?

Interests?



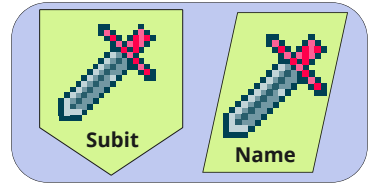
Any specific goals for the course?



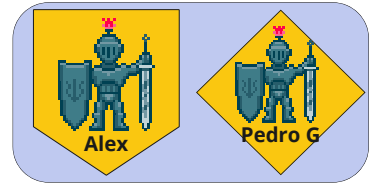
Learners can be optionally grouped into teams with breakout rooms.
How many Monty Python references? Count them during the course!

Choose a Quest Avatar or create your own Python themed one

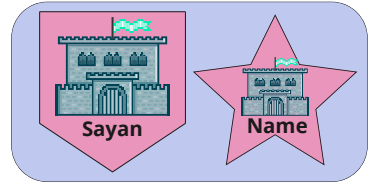
Excalibur



Brave Knights

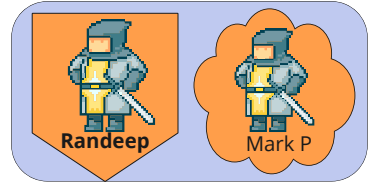


Swamp Castle



European Swallows

Peasants



Brave Horses

Questions and comments

Questions for tutor and open to all students who go here..

To do | 10	In progress | 0	Done | 0
1.		
2.		
3.		
4.		
5.		
6.		
7.		
8.		
9.		
10.		

Code and links shared by tutor

Python links in here please

<Code shared by tutor will go here...

Link for Project Starter Files
QACS-TL/Python_TL_Lab_Starter_Files

Link for Board Python Demos
QACS-TL/Python_TL_Demos

Link for lab solutions:
QACS-TL/Python_TL_Lab_Sols

Code shared by learner

<Code shared by learner will go here...

Python is a versatile programming language, capable of building virtually any type of application. Its extensibility is one of its greatest strengths, allowing developers to enhance core functionality with thousands of library modules. These modules empower us to achieve exciting and complex tasks with just a few lines of code. In the professional world, Python excels in areas such as data analysis, artificial intelligence, scientific computing, backend web development, and even automating system tasks and parsing log files.

We live in a data-driven world where the internet connects us, drives commerce, and provides entertainment. Data influences countless aspects of our lives, making it increasingly difficult to perform our jobs or manage daily tasks without some understanding of programming. Python is celebrated for its ease of learning and use, enabling developers to create impressive applications with minimal code – often by reusing existing code through library modules. In fact, reusability is often considered the Holy Grail of software engineering.

In the QA Live session, we'll embark on a quest to explore the exciting possibilities Python offers. Our primary goal is to have fun while learning, so we'll write programs and modules that download interesting data directly from the web, including weather information, police crime data, and top movie rankings. We'll then display, filter, graph, and save this data to files and databases – tasks that mirror real-world applications. Through this journey, we hope to inspire you to recognise the wealth of reusable resources available and apply them to your own projects.

Goals for our quest!

Reinforce key learnings: Solidify the concepts you learned in the DIGITAL session.
Program structure: Demonstrate how to effectively structure a Python program and its modules.
Namespaces and imports: Explore the proper use of namespaces and how to import modules correctly.
Data structures: Show how to utilise the appropriate data structures and objects (collections) to store and manage data.
Regular Expressions: Demonstrate how to match data accurately using Regular Expressions.
Filtering data: Explore methods for filtering collections using `filter()`, lambda functions, and list comprehensions.
File system interaction: Demonstrate how to interact with the file system by reading, writing, and appending data sequentially.
Database connectivity: Show how to connect to a SQLite database, create tables, populate them, and execute queries.
Object-Oriented Programming (OOP): Investigate OOP principles and their benefits.
Class and object creation: Demonstrate how to create user-defined classes, objects, and methods.
OOP concepts: Explore key OOP concepts such as Inheritance, Encapsulation, Polymorphism, duck typing, and operator overloading.
Advanced OOP techniques: Examine getter and setter methods, properties, and decorators.
Standard libraries and beyond: Investigate some of Python's Standard Libraries and external libraries to interact with the outside world.

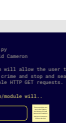

```
#!/usr/bin/env python
# Name:         menu.py
# Author:       QA2 A. Donald Cameron
# Revision:     v1.0
# Description: A fun and interactive demo program for QA2.0 Python LIVE.

"""
    A fun program that allows users to choose from a variety of
    interesting options, including accessing the latest weather
    reports, crime data, and top movies. The program demonstrates
    Python skills such as web scraping, API utilisation, and
    basic SQLite database interactions.
"""

import sys
import police_data
import weather_data
import movies

# Displayed menu options
MENU_TOP = """
    Fun Data Menu

    1. Display Online Public Weather Data in Browser
    2. Display Online Public UK Police Data to Screen
    3. Display/Search Top Movies from Letterboxd
    Q. Quit
"""

def display_menu():
    """Displays the top-level menu and handles user input."""
    while True:
        print(MENU_TOP)
        option = input("Enter an option (1-3, Q to quit): ").strip().lower()

        if option == "1":
            weather_data.weather_menu()
        elif option == "2":
            police_data.police_menu()
        elif option == "3":
            movies.menu()
        elif option == "q":
            print("Quitting...")
            break
        else:
            print("Invalid option, please try again.")
    return None

def main():
    """Entry point for the program. Initiates the top-level menu."""
    display_menu()
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

```python
#! /bin/python
# Name:        weather_data.py
# Author:      QA2.0, Donald Cameron
# Revision:    v1.0
# Description: Prompt user for city/country and then will get the online weather
#              from openweather.com, and the latest police incidents from
#              and either: display info in browser tab, save to DB or save to excel file.
#
# Requires:    This program requires a free API key to access online weather

"""
    Get, display and save weather data for a city.
"""

# Required libraries.
import sys
import re  # Provides regexp support
import webbrowser  # Provides functions for opening and displaying html content to web browsers.
import requests
import weather_db
import datetime

MENU_WEATHER = """
    Weather Data
    ------------
    1. Get online weather data for city, country
    2. Display weather data to web browser
    3. Save weather data to local database
    4. Query all rows in local database
    5. Delete row in local database
    6. Delete all rows in local database
    7. Drop local database table

"""

# 8-wind compass rose
compass = ["North", "North East", "East", "South East", "South", "South West",
           "West", "North West", "North"]

# API KEY ONLY TO BE USED IN CLASSROOM DEMONSTRATIONS!
# PLEASE subscribe at https://openweathermap.org/api for your own API-KEY.
# API_KEY = os.environ.get("OPEN_WEATHER_API")
API_KEY= "396a2d3386e07f92777ed7fee73f7c67"   # For testing... and to be removed.


def get_weather():
    """ Get weather for a given city, country and return dict """

    city = input("Please enter a city (eg. london): ") or "london"
    country = input("Please enter a country (eg. gb/uk): ") or "gb"

    base_url = r"http://api.openweathermap.org/data/2.5/weather?"
    full_url = f"{base_url}q={city},{country},&appid={API_KEY}"

    try:
        weather_data = requests.get(full_url)
        w_data = weather_data.json()
    except Exception as err:
        # if there was an error opening the url then return error
        print(f"Error: {err.args}", file=sys.stderr)
        return "Error opening url"
    else:
        # if weather_data has message 'city not found' from web server then city is invalid.
        if re.search(r"city not found", weather_data.text):
            print("City Not found", file=sys.stderr)
            return None

    # return the weather condition as a dict.
    return w_data


def display_weather(forecast:'dict')->'None':
    """ Format and display weather information to a browser tab """
    # Parse fields from forecast dictionary.
    city = forecast["name"]
    country = forecast["sys"]["country"]
    desc = forecast["weather"][0]["description"]
    icon = forecast["weather"][0]["icon"]
    icon_url = f"http://openweathermap.org/img/wn/{icon}@2x.png"
    temp = forecast["main"]["temp"] - 273.15 # Convert Kelvin to Celsius.
    humidity = forecast["main"]["humidity"]
    speed = forecast["wind"]["speed"]
    compass_idx = round(forecast["wind"]["deg"] // 45)
    direction = compass[compass_idx]
    feels_like = forecast["main"]["feels_like"] - 273.15 # Convert Kelvin to Celsius.
    now = datetime.datetime.now()
    time = now.strftime("%H:%M:%S")

    # Format fields into html string plus optional WIDGET
    cel_sign = "\N{degree sign}C"
    outlook = f"""
        <html>
           <head><title>Weather for {city}</title></head>
           <body><p><a>  Weather for {city}, {country} at {time}</a></p></body>
           <body><p><tr> Weather is {desc} <IMG SRC="{icon_url}" ALT="Weather" WIDTH=64 HEIGHT=64> </tr></p></body>
           <body><p>     Temp is {temp:.0f}{cel_sign} but feels like {feels_like:.0f}{cel_sign} </p></body>
           <body><p>     Wind is {speed} towards {direction}</p></body>
           <body><p>     Humidity is {humidity} </p></body>
           <body></body>
        </html>
    """

    # write forecast to html file
    try:
        with open('weather.html', 'w') as f_html:
            f_html.write(outlook)
    except Exception as err:
        print(f"Error: {err.args}", file=sys.stderr)
        return None

    # display weather forecast to new web browser tab
    try:
        webbrowser.open_new_tab("weather.html")
    except webbrowser.Error as err:
        print(err, file=sys.stderr)
    return None
```

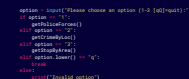
Format HTML code and write to file.

This will open a browser tab with the contents of weather.html file

```python
def weather_menu():
    """ Menu for Weather Data """
    data = weather_db.Weather_db()  # Instantiate DB object
    city_w = {}
    while True:
        print(MENU_WEATHER)
        option = input("Enter an option (1-7, [qQ=quit]): ")
        if option == "1":
            city_w = get_weather()
        elif option == "2":
            if city_w:
                display_weather(city_w)
            else:
                print("No weather downloaded")
        elif option == "3":
            data.create_table()
            data.insert_row(table_name="weather", city=f"{city_w['name']}"
                            , country=f"{city_w['sys']['country']}"
                            , wind_direction=f"{city_w['wind']['deg']}"
                            , temp=f"{city_w['main']['temp']}")
            data.commit()
        elif option == "4":
            data.query_all()
        elif option == "5":
            row = input("Choose a RowID to delete? ")
            data.delete_row(row)
            data.commit()
        elif option == "6":
            data.delete_all_rows()
            data.commit()
        elif option == "7":
            data.drop_table()
        elif option.lower() == "q":
            print("Quitting..")
            break
        else:
            print("Invalid option")

    return None

def main():
    """ This module can be run directly to retrieve weather data """
    weather_menu()
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```



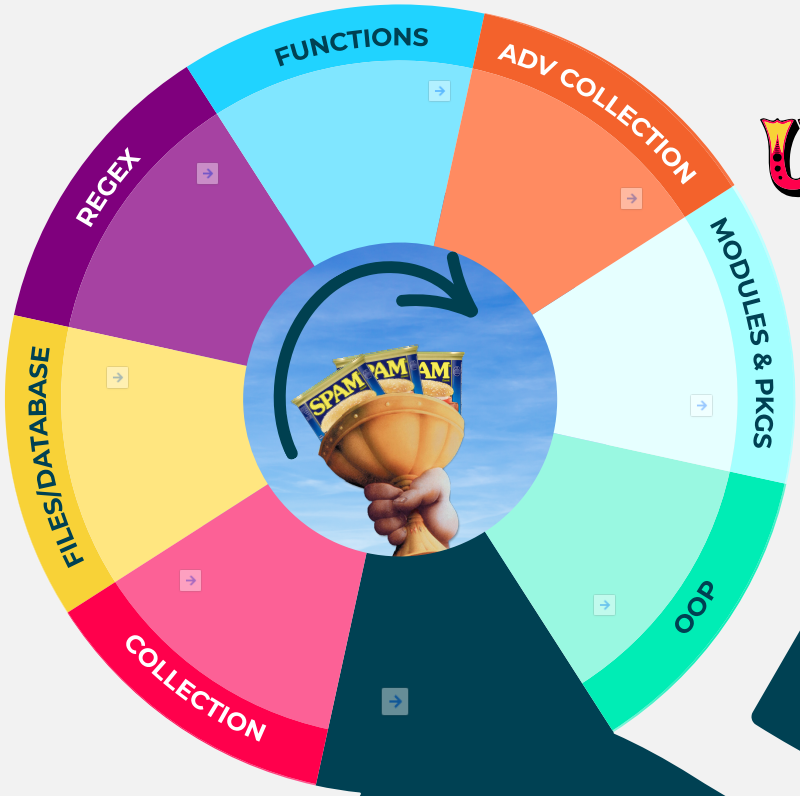
```
#!/bin/python3
# Name:         police_data.py
# Author:       Gal.B, Donald Cameron
# Revision:     v1.0
# Description:  This program will allow the user to retrieve publicly
# available Police Forces, crime and stop and search information for
# specific areas using simple HTTP GET requests.
"""
    Docstring: This program/module will..
"""
import webbrowser
import requests
import os
import pprint

app_name = "Welcome to POCA - the Police Data Crime App"
MENU_POLICE = """
    Police Data
    -----------
    1. List of Police Forces
    2. Crimes by location
    3. Stop and Search by Location

"""
base_url = r"https://data.police.uk/api/"

def getPoliceForces():
    """ Display List of Police forces """
    try:
        full_url = base_url + "forces"
        force_data = requests.get(full_url) # HTTP GET url page.

        for force in force_data.json():
            print(f'{force[ id ]:<10s}: {force[ name ]:<20s}')
        print()
    except Exception as err:
        print(f'Error, cannot retrieve info: {{err}}', file=sys.stderr)
    return None

def getCrimeByLoc():
    """ Display Police Crime data for a specific lat/long """
    date = input('Enter a date (YYYY-MM): ')
    location = input('Enter a Lat,Long (eg. 51.506,-0.075 [London Bridge]): ')
    if not location:
        lat = '51.5Nn'
        lon = '-0.075'
    else:
        lat, lon = location.split(',')

    try:
        full_url = base_url + 'crimes-at-location?lat=' + lat + '&lng=' + lon + '&date=' + date
        crime_data = requests.get(full_url)
    except Exception as err:
        print(f'Error, {err.args}, file=sys.stderr)
        return None
    else:
        for crime in crime_data.json():
            print(f'{crime[ category ]}, {crime[ location_type ]}, '
                  f'{crime[ outcome_status ][ category ] if crime[ outcome_status ] else  No Investigation' }, '
                  f'{crime[ location ][ street ][ name ]}')
        print()

    return None

def getStopByArea():
    """ Display Police Stop Data for a specific lat/long """
    date = input('Enter a date (YYYY-MM): ')
    location = input('Enter a Lat,Long (eg. 51.506,-0.075 [London Bridge]): ')
    if not location:
        lat = '51.5Nn'
        lon = '-0.075'
    else:
        lat, lon = location.split(',')

    try:
        full_url = base_url + 'stops-street?lat=' + lat + '&lng=' + lon + '&date=' + date
        stops_data = requests.get(full_url)
    except Exception as err:
        print(f'Error {err.args}', file=sys.stderr)
        return None
    else:
        for crime in stops_data.json():
            print(f'{crime[ object_of_search ]:s} '
                  f'{crime[ gender ]} ({crime[ age_range ]}), {crime[ type ]}, '
                  f'{crime[ location ][ street ][ name ]}')
        print()
    finally:
        print()

    return None

def police_menu():
    """ Menu for Police Crime Data """
    while True:
        print('\n{app_name}')
        print('-' * len(app_name))
        print(MENU_POLICE)

        option = input('Please choose an option [1-3 [q]=quit]:')
        if option == '1':
            getPoliceForces()
        elif option == '2':
            getCrimeByLoc()
        elif option == '3':
            getStopByArea()
        elif option.lower() == 'q':
            break
        else:
            print('Invalid option')

    return None

def main():
    """ This module can be run directly to retrieve police crime data """
    police_menu()
    return None

if __name__ == '__main__':
    main()
    sys.exit(0)
```



```python
#!/bin/python
# Name:         weather_db.py
# Author:       Sir A. Donald Cameron
# Revision:     v1.0
# Description: This module defines a Weather class for storing data
# in a sqlite3 database.
"""
    This module describes a Class of Weather objects which can create/drop tables
    and insert/delete rows in a weather database.
"""
import sys
import sqlite3
import pprint

class Weather_db:
    # Class variable storing location of DB
    # Could store this in an external file or environment variable.
    DB_LOC = "C:\labs\weather_db.sqlite"

    def __init__(self):
        try:
            self.__db_conn = sqlite3.connect(Weather_db.DB_LOC)
            self.cur = self.__db_conn.cursor()
        except Exception as err:
            raise ValueError
        # No return as not explicitly called.


    def create_table(self, table_name='weather'):
        self.cur.execute(f""" CREATE TABLE IF NOT EXISTS {table_name}
                            (id          INTEGER PRIMARY KEY
                            ,city        VARCHAR(30)
                            ,country     VARCHAR(30)
                            ,date        DATE DEFAULT CURRENT_DATE
                            ,description VARCHAR(50)
                            ,temp        FLOAT
                            ,humidity    FLOAT
                            ,wind_speed FLOAT
                            ,wind_direction VARCHAR(2)
                            ) """)
        return None

    def drop_table(self, table_name='weather'):
        """ Drop table if exists """
        option = input(f'Are you sure you want to drop {table_name} (y/n)? ')
        if option.lower() == 'y':
            self.cur.execute(f""" DROP TABLE IF EXISTS {table_name}""")
        return None

    def query_all(self, table_name='weather'):
        """ Query and return all rows in formatted str output """
        sql = f'SELECT * FROM {table_name}'
        try:
            self.cur.execute(sql)
            rows = self.cur.fetchall()
            for row in rows:
                print(row)
        except Exception as err:
            print(f'Error accessing {table_name}: {err}\n', file=sys.stderr)

        return None

    def insert_row(self, table_name='weather', **kwargs):
        """ Insert new row into weather db table """
        weather = {'id': None, 'city': None, 'country': None,
                   'date': 'DEFAULT', 'description': 'N/A', 'temp': None,
                   'humidity': None, 'wind_speed': None, 'wind_direction': None}
        weather.update(kwargs)
        pprint.pprint(weather)
        sql = f'INSERT INTO {table_name} VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)'

        try:
            self.cur.execute(sql, (weather['id'], weather['city'], weather['country'],
                                   weather['date'], weather['description'], weather['temp'],
                                   weather['humidity'], weather['wind_speed'], weather['wind_direction'] ))
        except Exception as err:
            print(f'Cannot insert row, {err}', file=sys.stderr)

        return None

    def delete_row(self, row_id, table_name='weather'):
        """ Delete new row into weather db table """
        try:
            sql = f'DELETE FROM {table_name} WHERE id={row_id}'
            self.cur.execute(sql)
        except Exception as err:
            print(f'Error accessing {table_name}: {err}\n', file=sys.stderr)
        return None

    def delete_all_rows(self, table_name='weather'):
        """ Delete all rows in the the table."""
        try:
            sql = f'DELETE FROM {table_name}'
            self.cur.execute(sql)
        except Exception as err:
            print(f'Error accessing {table_name}: {err}\n', file=sys.stderr)
        return None

    def commit(self):
        """ Commit changes to database """
        self.__db_conn.commit()
        return None

    def close(self):
        self.__db_conn.close()
        return None

# Example of operator overloading.
    def __str__(self):
        """ Return status of DB connection a str """
        if self.__db_conn:
            return f'DB still connected'
        else:
            return f'DB not connected'

    def __del__(self):
        """ Close connection automatically when object is deleted """
        self.__db_conn.close()
        return None
```




```
#! /bin/python
# Name:          movies.py
# Author:        QA2.0, Donald Cameron
# Revision:      v2.0
# Description: This program will download the top 250 movies from Letterboxd
# and will allow the user to display the top-n ranked movies or search
# for their favourite movies.

    Download and display the Top 250 Movies Data from Letterboxd.
"""
import sys
import requests
from bs4 import BeautifulSoup

import re

MENU_MOVIES = """
    Movies Menu
    -----------
    1. Get online movie ranking from Letterboxd
    2. Display top ranking movies
    3. Search for movie
    Q. Quit
"""

def get_movies():
    """ Web Scrape the top 250 movies online from letterboxd.com """
    # Base URL of the Letterboxd Top 250 movies page
    base_url = 'https://letterboxd.com/jack/list/official-top-250-films-with-the-most-fans/page/{}/'
    top_movies = {} # Movie info - 'title': [rank, rating, img_url]

    # Scrape the first 3 pages (adjust if necessary)
    for page_num in range(1, 4):  # Adjust range according to the number of pages
        url = base_url.format(page_num) # Send a GET request to fetch the page content
        response = requests.get(url)
        soup = BeautifulSoup(response.text, 'html.parser')

        # Find all movie containers
        movie_tags = soup.find_all("li", class_="poster-container")

        # Extract movie details
        for rank, movie in enumerate(movie_tags, start=1 + len(top_movies)):
            title = movie.find('img', class_='image').get('alt')
            link = movie.find('div', class_='really-lazy-load').get('data-target-link')
            rating = movie.get('data-owner-rating')
            full_link = f"https://letterboxd.com{link}"

            top_movies[title] = [rank, rating, full_link]
    return top_movies

def display_movies(movies, top=250):
    """ Display top movies in a given dict """
    if not movies:
        print("No movies to display.")
    else:
        for movie, info in movies.items():
            rank, rating, link = info
            print(f'{rank:>4d} {movie:<s}, {rating}/10 - {link}')
            if int(rank) >= int(top): break
    return None

def search_movies(pattern='r'.*, movies=None):
    """ Search for pattern in a given dict of movies """
    if not movies:
        print("No movies to search.")
    else:
        movies_matched = 0
        for movie, info in movies.items():
            rank, rating, link = info
            if re.search(pattern, movie, re.IGNORECASE):
                print(f'{rank:>4} {movie:<s}, {rating}/10 - {link}')
                movies_matched += 1
        print(f'{movies_matched} movies matched')
    return None

def menu():
    """ Movie Menu """
    films = {}
    while True:
        print(MENU_MOVIES)
        option = input("Enter option (1-3, [q=quit]): ").strip().lower()

        if option == '1':
            films = get_movies()
            print(f"Fetched {len(films)} movies")
        elif option == '2':
            top = input("How many top movies do you want to display [default=250]: ").strip() or "250"
            display_movies(films, top)
        elif option == '3':
            pattern = input("Enter title search pattern: ").rstrip()
            search_movies(pattern, films)
        elif option == 'q':
            print("Exiting...")
            break
        else:
            print("Invalid option, please try again")

    return None

if __name__ == "__main__":
    menu()
    sys.exit(0)
```



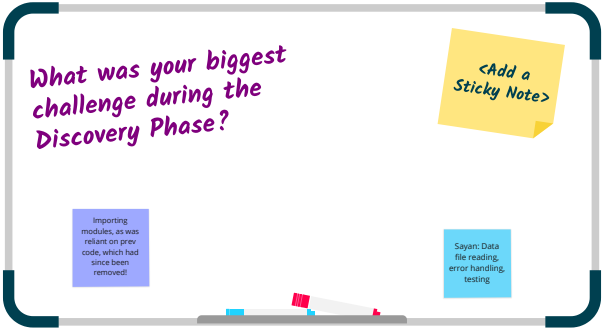
UEST PPLY

STARTS HERE

Digital Phase Activities

These activities should have been completed as part of the Digital Phase of your learning. Most will be covered again in greater detail during the Live phase.

Course Introduction	if, elif and else	len(), min(), max(0) and sum()
Introduction to Python 3	Indentation and 4 spaces	Mutable and immutable
IDLE & REPL prompt	Comparison operators	Repetition using while and for loops
Variables & objects	Boolean and 'in' operators	Introduction to file handling and 'with'
Comments	Unicode and special chars	Reading and writing text files
PEP 8 Style Guide	str indexing and slicing	Reading and writing csv files
keyboard inpu()	str concatenation and repetition	Reading and writing to DB files
int, float, str, list and booleans	str methods	Introduction to user functions
type(), dir() and help()	str formatting using f-strings	Introduction to Python Std Library
Casting/conversion	Collections introduction	Importing Modules
Math and string operators	Tuples, lists, dict and sets	Introduction to Unit Testing
Flow control/decision making	Collection indexing and slicing	Introduction to Error Handling



Mini Lab

Refresh digital concepts before main Quest

20 mins

During your Digital phase, you covered topics on variables, data types, control flow, loops, and simple user functions. In this series of mini-labs, you will review your knowledge of these topics.

1

Lab 1: Variables and Data Types

1. Create four variables to hold your name, age, height and whether you are a student (hint: boolean).
2. Print all the variables.
3. Use the type function to display the data type of each variable.

2

Lab 2: Control Flow (if-else statements)

1. Prompt the user to enter a number and check if it is positive, negative, or zero.
2. Prompt the user for a year and check if it is a leap year.

3

Lab 3: Loops

1. Write a for loop that prints numbers from 1 to 10.
2. Write a while loop that calculates the sum of numbers from 1 to 100.

4

Lab 4: Functions

1. Write a function that takes two numbers as arguments and returns their sum.
2. Write a function that checks if a given string is a palindrome (reads the same forwards as backwards).

5

Lab 5: Formatting Strings

1. Create variables to hold your name, age, height and country, and use a f-string to print a sentence that includes all this information.

Mini Lab Solution/s

Refresh digital concepts before main Quest

Move box
to reveal answers

Move box
to reveal answers

Move box
to reveal answers

Move box
to reveal answers

Move box
to reveal answers

Move box
to reveal answers

Move box
to reveal answers

Move box
to reveal answers

Questions...

Now is your chance to clear up loose ends from the Digital Phase

Short Question & Review

<Add a Sticky Note>

PyCharm

Let's get ready to code

Download Community Edition from here:
https://www.jetbrains.com/pycharm/download/#section=windows
Documentation for PyCharm can be found here:
https://www.jetbrains.com/help/pycharm/quick-start-guide.html

PyCharm is a dedicated Python IDE providing more functionality, tools, and a more engaging experience for developers, including project management, virtual environments, and GitHub integration.

Three editions

- Community – free
- Professional (data science and version control)
- Edu (learning)

Multi-platform

Cool features

- Python shell command line with colour coded highlighting and error messages
- Multi-tab text editor with colour coded highlighting, auto indentation and completion
- Search and replace tool
- Powerful debugger with breakpoints
- Support virtual environments
- Support Projects and Packages
- Integrated support for GitHub, Mercurial, and Subversion, etc.

PyCharm

Let's get ready to code

Mostly everything you do in **PyCharm** revolves around **Projects**.
They act as a basis for grouping related code, allowing refactoring/renaming within the group, better management of modules, etc.

Start Pycharm CE (Community Edition)

Create a new Project and choose folder location inside **C:\labs.**

Select VirtualEnv to create an isolated install of Python with its own Interpreter and modules.

Choose a Base Interpreter and select Inherit Global Site Packages box.

File>**Settings**

File>Settings>Editor>**Font**

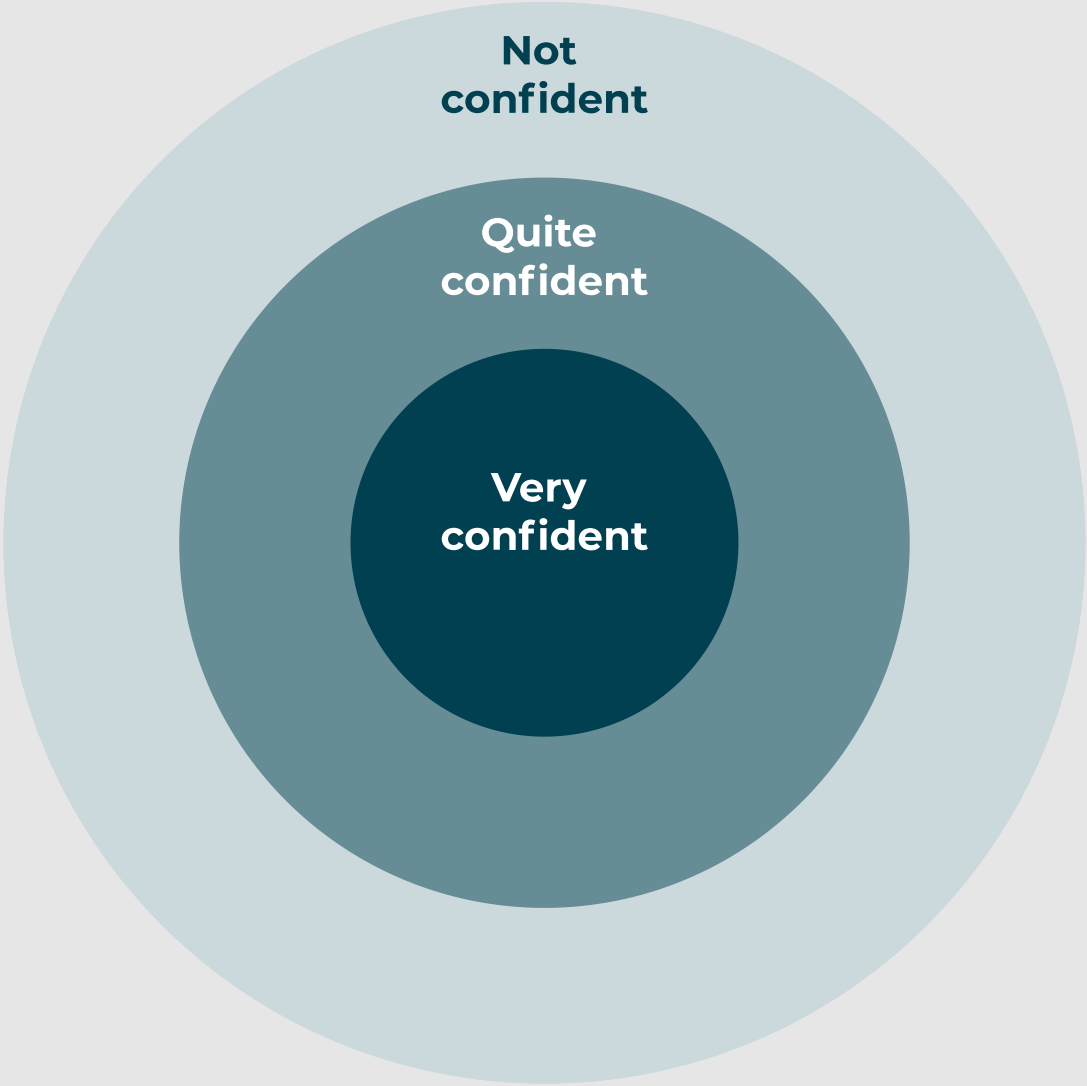
File>Settings>Editor>Font>**Color Scheme**

File>Settings>Editor>Font>File and Code Templates>**Python**

Review Navigation Bar and Editor.

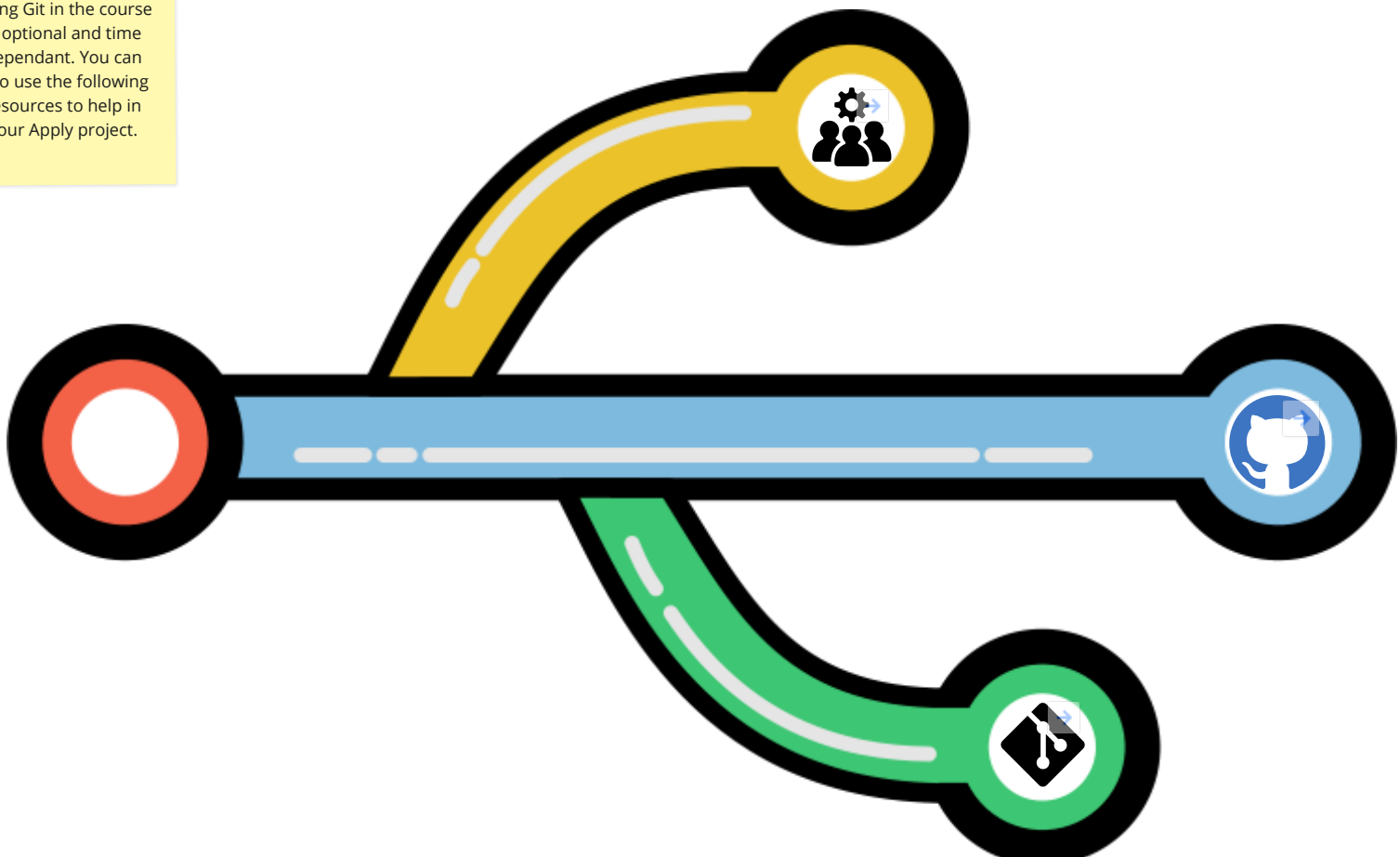
Project>New>**Python File**

Compile and run script. 



Back to the quest

Using Git in the course is optional and time dependant. You can also use the following resources to help in your Apply project.



Collections

Storing things in an ordered or unordered way

Where do I put all these things?

Tuples LISTS DICTIONARIES SETS

COLLECTIONS

In Python, tuples, lists, dictionaries, and sets are built-in data structures that serve different purposes. They are used to store and organise collections of data, each with its own characteristics, strengths, and typical use cases.

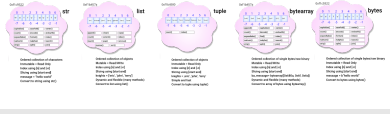
These data structures can be categorised into **Ordered** and **Unordered**.

Strings, tuples, and dictionaries are ordered, and dictionaries* and sets are viewed as unordered.

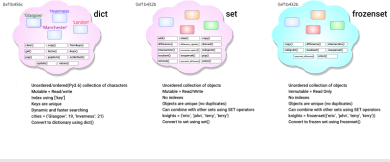
*Since Python 3.6, dictionaries are stored in Insertion Order.

We can visualise the key characteristics of tuples, lists, dictionaries, and sets. There are other types of collections including bytesarray, bytes, and frozensets but these will not be demonstrated further. Use help() or ask your instructor for more details.

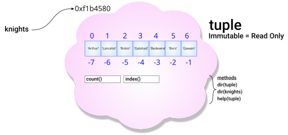
Ordered Collections



Unordered Collections



TUPLES



Purpose:

- Tuples are used to store ordered, immutable sequences of elements. Once created, the elements of a tuple cannot be changed, added, or removed.

Key characteristics:

- **Immutable:** The elements cannot be modified after creation.
- **Ordered:** Elements have a specific order, and they can be accessed via indexing.
- **Heterogeneous:** Can store elements of different data types (e.g., integers, strings, other tuples).
- **Hashable:** Tuples can be used as keys in dictionaries if they contain only hashable elements.

Use cases:

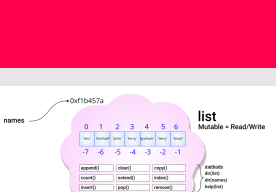
- Storing data that should not change, such as fixed configurations.
- Returning multiple values from a function.
- Grouping related items together.

Try these statements at the Python shell and explain what is happening?

```
>>> a = 10
>>> b = a
>>> (a, b) = (b, a)
>>> print(a)
>>> print(b)
>>> (a, b, c) = range(0, 3)
>>> print(a, b, c)
>>> (a, b, c) = (10, 20, 30, 40, 50)
>>> (a, b, *c) = (10, 20, 30, 40, 50) # This is called unpacking and is useful when LHS and RHS are not in balance.
>>> print(a, b)
>>> print(c)
>>> comment = ("Tis but a scratch")
>>> print(type(comment))
>>> comment = ("Tis but a scratch", )
>>> print(type(comment))
```

```python
#! /bin/python
# Name:        demo_tuples.py
# Author:      QA2.0, Donald Cameron
# Revision:    v1.0
# Description: This program will demonstrate how to create and access a tuple
# which is an Immutable (Read-only) Ordered Collection of objects.
"""
    Creating, indexing and accessing objects within a Tuple.
"""
import sys

def main():
    """   """
    scottish_bands = ("The Proclaimers", "Franz Ferdinand", "Primal Scream"
                      , "Simple Minds")

    print(f"Scottish bands is of {type(scottish_bands)}")
    print(f"There are {len(scottish_bands)} bands in {scottish_bands}.")

    # We can Index[] and Slice [start:end] a tuple.
    print(f"The 1st band is {scottish_bands[0]}")
    print(f"The last band is {scottish_bands[-1]}")
    print(f"First three bands = {scottish_bands[0:3]}")

    try:
        # But cannot modify a Tuple as it is immutable.
        # Try this without the try/except block.
        scottish_bands[2] = "Deacon Blue"
    except TypeError:
        print("Cannot modify a Tuple")

    print(f"Primal Scream exists {scottish_bands.count('Primal Scream')} "
          f"at index {scottish_bands.index('Primal Scream')}")
    return None


if __name__ == "__main__":
    main()
    sys.exit(0)
```

LISTS



Purpose:

- Lists are used to store ordered, mutable sequences of elements. They are versatile and commonly used to maintain collections of items that may change in size or content.

Key characteristics:

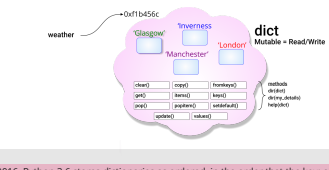
- **Mutable:** Elements can be modified, added, or removed after creation.
- **Ordered:** Elements have a specific order, and they can be accessed and modified via indexing.
- **Heterogeneous:** Can store elements of different data types.
- **Dynamic Size:** Lists can grow or shrink as elements are added or removed.

Use cases:

- Maintaining a collection of items where the order matters, and where the collection needs to be modified (e.g., a list of tasks).
- Performing operations that involve frequent updates or access by index.

```
#! /bin/python
# Name:         demo_lists.py
# Author:       GAZ.@, Donald Cameron
# Revision:     v1.8
# Description:  This program will demonstrate how to create, grow, shrink and access
# objects within a list.
"""
    Playing with a list of numeric (numbers) and non-numeric (movie titles) data.
"""
import sys

def main():
    """ Create, grow, shrink and play with lists (numeric and non-numeric) """
    numbers = [256, 32, 46, 9, 31, 79, 122]
    movies = ['Local Hero', 'Gregory's Girl', 'Shallow Grave']

    # A few cool built-in functions for finding, length, smallest
    # largest and sum of numbers.
    print(f'Length of numbers = {len(numbers)}')
    print(f'Smallest number = {min(numbers)}')
    print(f'Largest number = {max(numbers)}')
    print(f'Sum of numbers = {sum(numbers)}')

    # Some of these functions (Not sum) can also be used on non-numeric data.
    print(f'Number of movies = {len(movies)}')
    print(f'1st movie in char order = {min(movies)}')
    print(f'Last movie in char order = {max(movies)}')

    # We can Index and Slice [start:end] lists.
    print(f'1st movie = {movies[0]}')
    print(f'Last movie = {movies[-1]}')
    print(f'Last three movies = {movies[-3:]}')

    # Let's try some list methods for adding to the lhs, rhs
    # and within the list. List will be re-indexed if required.
    movies.append('Trainspotting') # Append one object to RHS.
    movies.extend(['Brave', 'Braveheart']) # Extend multiple objects to RHS.
    movies.insert(0, 'Comfort & Joy') # Insert one object to LHS (before index 0).
    movies.insert(1, 'The Angel's Share') # Insert on object before index 2.
    print(f'Scottish movies = {movies}')

    # Let's try some list methods for removing from lhs, rhs
    # and within the list. List will be re-indexed if required.
    last = movies.pop() # Last object removed and optionally returned.
    first = movies.pop(0) # First object removed and optionally returned.
    movies.pop(3) # Remove object at index 3.
    print(f'Scottish movies = {movies}')

    # movies.clear() # Empties list.
    # films = movies.copy() # Copies List.
    # Notice how print parameters can span across lines.
    print(f'I spotted Trainspotting {movies.count('Trainspotting')} '
          f'at index {movies.index('Trainspotting')}')

    # The built-in sorted function returns a sorted list.
    print(f'Movies sorted = {sorted(movies, key=str, reverse=False)}')
    print(f'Movies sorted = {sorted(movies, key=str, reverse=True)}')
    print(f'Movies sorted = {sorted(movies, key=len)}') # Bespoke sort.

    movies.sort() # Sorts list in place.
    movies.reverse() # Reverses list in place.
    movies.remove('Trainspotting')

    print(f'List of movies = {movies}') # Saturday night viewing.
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

DICTIONARIES



Since 2016, Python 3.6 stores dictionaries as ordered, in the order that the key: objects are assigned. This may change in a future implementation, so don't assume!

Purpose:

- Dictionaries are used to store key-value pairs, where each key is unique and maps to a specific value. They are ideal for fast lookups by key.

Key characteristics:

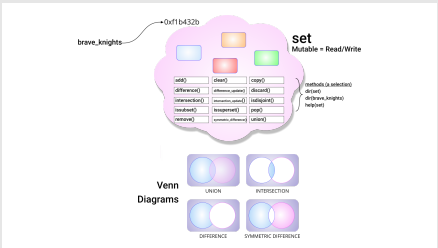
- **Mutable:** Values associated with keys can be modified, and new key-value pairs can be added or removed.
- **Unordered (until Python 3.7):** Keys do not have a specific order, though Python 3.7+ maintains insertion order.
- **Key-Value Pairs:** Each key maps to a corresponding value, and keys must be unique.
- **Keys are hashable:** Typically, keys are strings or numbers, but they can be any immutable type.

Use cases:

- Associating data with unique identifiers (e.g., mapping user IDs to user information).
- Performing fast lookups, additions, or deletions of elements by key.

```
#! /bin/python
# Name:         demo_dictionaries.py
# Author:       GAZ © Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to create, grow and shrink
# dictionaries and use keys and iterator for loops to access objects within.
# Dictionaries are mutable collections that can store multiple objects and
# access using UNIQUE keys.
"""
    Weather Program to store and display average (temp, precipitation and humidity)
    for UK cities in 2020.
"""
import sys
import pprint

def main():
    """ Display weather info for uk cities """

    # Create a dictionary to store annual average weather information
    # (Mean Temp, Precipitation, Humidity) for UK Cities in 2020.
    weather = {'Glasgow' : [5, 53.6, 87],
               'London' : [12, 40.7, 81],
               'Edinburgh' : [4, 37.6, 81],
               'Manchester' : [7, 35.6, 88],
               'Thurso' : [6,89.6, 85]
    }

    # Add two new key: objects to weather dictionary.
    weather['Southampton'] = [6, 28.6, 92]
    weather['Inverness'] = [6, 22.5, 82]

    # weather.clear() # Empty dictionary.
    # w_info = weather.copy() # Copy dictionary.
    weather.pop('Thurso', False) # Remove Thurso as not a City.
    # weather.popitem() # Remove next key:object found.

    print(f'Number of cities recorded = {len(weather)}')
    print(f'Weather info for Glasgow = {weather["Glasgow"]}')
    print(f'Average precipitation = {weather["Glasgow"][1]} mm')

    # ITERATE through the key and objects within the dict.
    # keys() returns a VIEW into next key.
    for city in weather.keys():
        print(f'{city} average = {weather[city]}')

    # ITERATE through the values within the dict.
    # values() returns a VIEW into the next value.
    for w_info in weather.values():
        print(f'{w_info}')

    # ITERATE through the (values, objects) within the dict.
    # items() returns a VIEW into the next pair of (key, value).
    for city, w_info in weather.items():
        print(f'{city} weather info = {w_info}')
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

SETS



Purpose:

- Sets are used to store unordered collections of unique elements. They are optimized for operations like membership tests, union, intersection, and difference.

Key characteristics:

- **Mutable (with an immutable counterpart called frozenset):** Elements can be added or removed.
- **Unordered:** Elements do not have a specific order.
- **Unique elements:** No duplicate elements are allowed.
- **Efficient membership tests:** Checking for the presence of an element is fast.

Use cases:

- Storing collections of items where duplicates are not allowed (e.g., unique tags).
- Performing mathematical set operations like union, intersection, and difference.
- Efficiently testing membership or eliminating duplicates from a collection.

There are many Brave Knights who would also like to be Lumberjacks and vice-versa, so we will attempt to combine sets of Knights and sets of Lumberjacks using common SET operators and methods.

```
#! /bin/python
# Name:         demo_dictionaries.py
# Author:       GKD, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to create, grow and shrink
# and combine sets using SET operators and methods.
# Sets are mutable collections that store multiple UNIQUE objects (with no keys).
"""
    The joy of SETS!
"""
import sys

def main():
    """ Combining collections of Brave Knights and Lumber Jacks """
    # Create two sets, one with objects and one empty.
    brave_knights = {'donald', 'brian', 'terry', 'eric', 'michael'}
    lumber_jacks = set() # Creates an empty set.

    # Sets are Mutable so can be grown (add) and shrunk(pop/remove).
    brave_knights.add('john')
    brave_knights.add('terry') # Sets automatically remove duplicates!
    lumber_jacks.add('donald')
    lumber_jacks.add('michael')
    lumber_jacks.add('graham')

    # brave_knights.pop() # Remove next object found.
    # brave_knights.remove('') # Remove object by name.

    # Sets can be combined using operators or methods. Remember Venn diagrams?
    # Combine SETS using SET methods.
    print(f'Brave Knights and Lumber Jacks = {brave_knights.union(lumber_jacks)}')
    print(f'Brave Lumber Knights  = {brave_knights.intersection(lumber_jacks)}')
    print(f'Strictly Brave Knights ONLY = {brave_knights.difference(lumber_jacks)}')
    print(f'Everyone BUT Brave Lumber Knights = {brave_knights.symmetric_difference(lumber_jacks)}')
    print('-' * 40)
    # Combine SETS using SET operators.
    print(f'Brave Knights and Lumber Jacks = {brave_knights | lumber_jacks}')
    print(f'Brave Lumber Knights = {brave_knights & lumber_jacks}')
    print(f'Strictly Brave Knights ONLY = {brave_knights - lumber_jacks}')
    print(f'Everyone BUT Brave Lumber Knights = {brave_knights ^ lumber_jacks}')
    return None

if __name__ == '__main__':
    main()
    sys.exit(0)
```

Quiz

Collections Knowledge Check

5-10 mins

1. Name three ordered and two unordered collections?

Ordered?

To be revealed

Unordered

To be revealed

2. Is a tuple mutable?

To be revealed

3. The joy of sets! What would be printed from the following code?

```
>>> scottish_movies = {"Gregory's Girl", "Local Hero", "Comfort and Joy", "That Sinking Feeling"}
>>> fav_movies = {"The Hobbit", "Local Hero", "The Lord of the Rings"}
>>> print(scottish_movies & fav_movies)
```

To be revealed

4. Fill in the blanks:

Dictionaries have unique To be revealed

Sets have unique To be revealed

5. What would be printed from the following code?

```
>>> movies = "Gregory's Girl", "Local Hero", "Comfort and Joy", "That Sinking Feeling"
>>> x, *y, z = movies
>>> print(y)
```

To be revealed

6. Write the print statement from question 3 using set methods?

To be revealed

Exercises

30 mins

Collections – exercises for 'brave' Knights

1

Write a Python script called **C:\labs\lotto.py** that will generate and display 6 unique random lottery numbers between 1 and 50. Think about which Python data structure is best suited to store the numbers! Use the Python help() function to find out which function to use from the python standard library called random.

2

Create a new script in **C:\labs** called **topTen.py**

a. Read the top 250 movies from the **C:\labs\top250_movies.txt** file and store them in a list called movies.
b. Print out the top 10 with rankings (right justified 5 characters) followed by a colon and space followed by name of the film.
c. Print out the first and last movie.
d. Modify the code to allow the user to choose the top N to be displayed.

Collections – stretch exercises for Kings

3

Create a reusable user function called **get_movies** which returns all the movie data into a list called movies.

4

Create another function called **display_movies** which accepts two parameters - a list of movies, and number of movies to be displayed.

5

Create another function called **search_movies** which accepts a simple string (to search), and the list of movies to search.

6

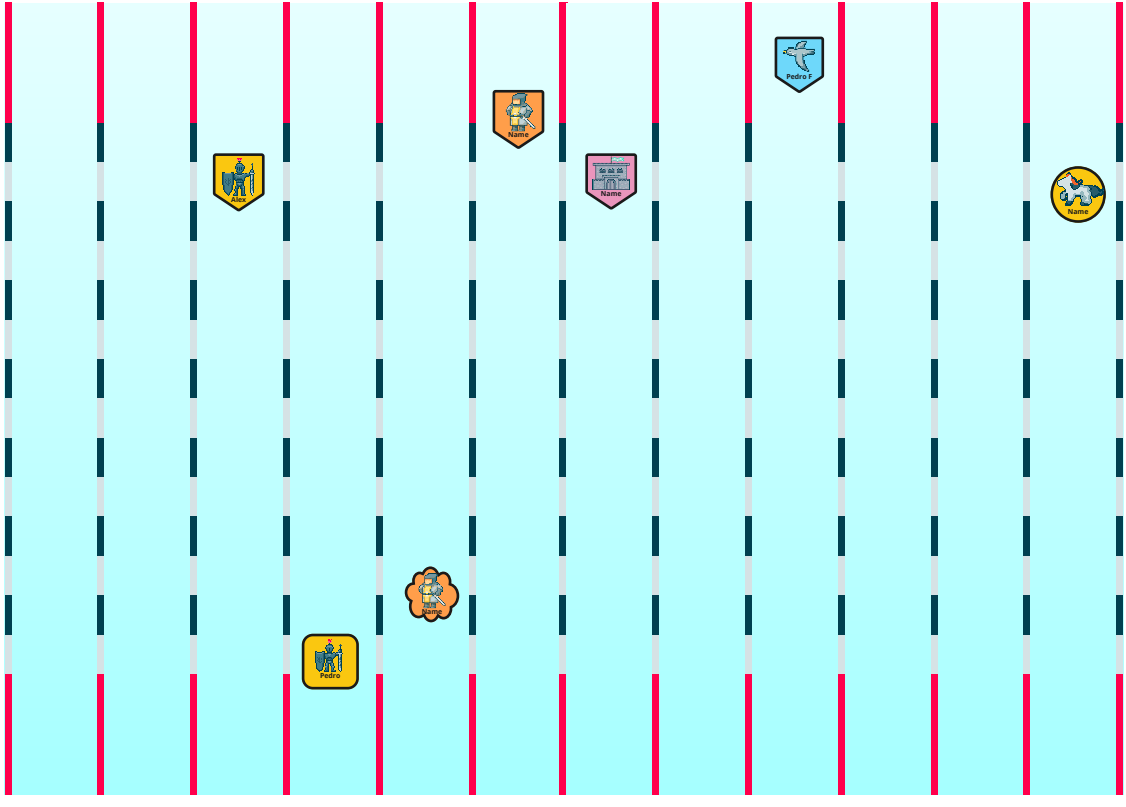
Now create a simple user menu which prompts the user for the following:

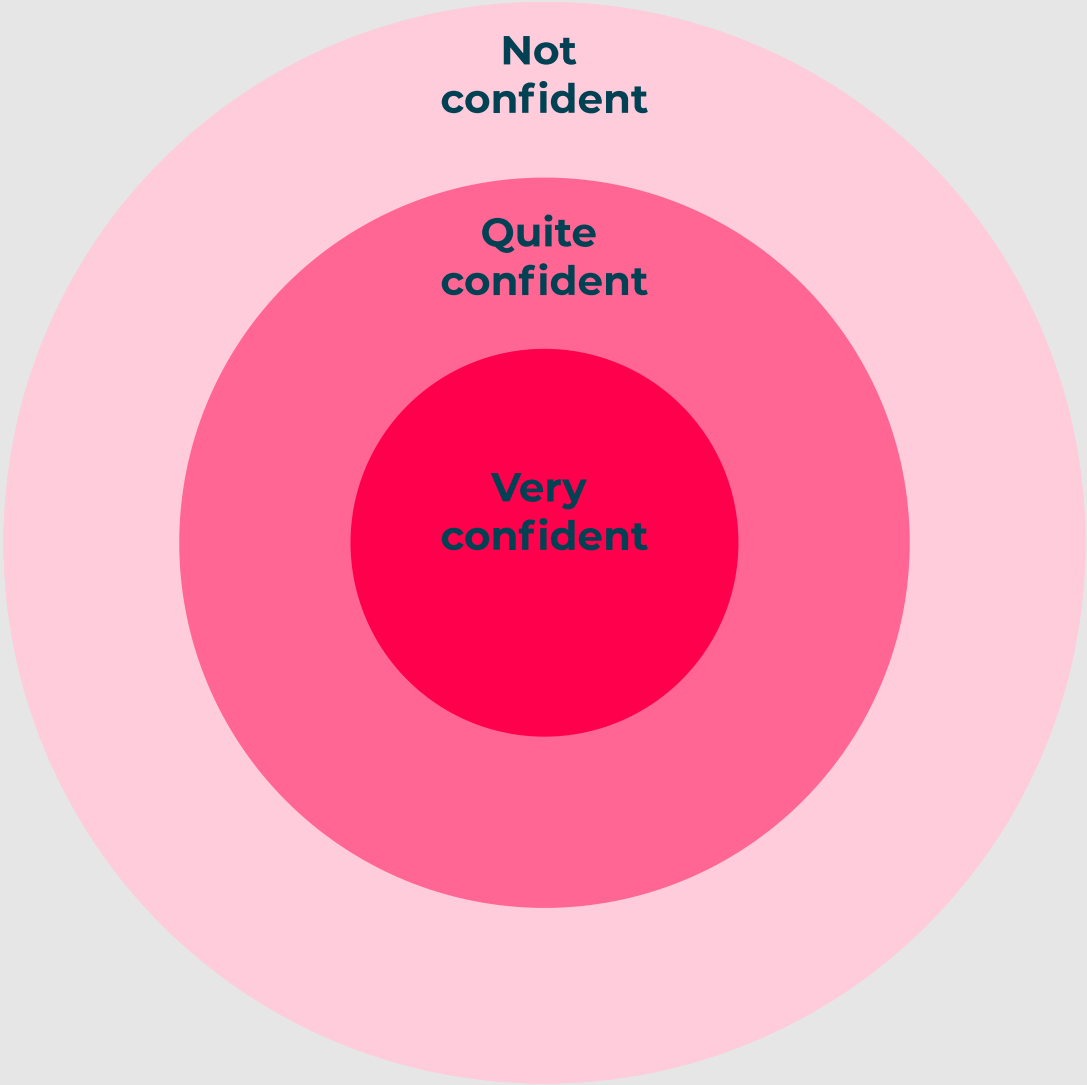
```
Movies Menu
-----------------
1. Get movie data from file
2. Display top ranking movies
3. Search for movie
Enter option (1-3, [qQ=quit]):
```

7

Extra kudos for having a main function and using the __name__ == "__main__" namespace trick.

Move your piece down as you complete the exercises





Back to the quest

Persistent Data

The Art of saving data to non-volatile storage

File Interaction Sequential Access

```
#! /bin/python
# Name:         demo_sequential_rw.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to Interact with
    the file
# system by opening/closing file handles for read, write and
    appending.
"""
    Write and read movie data to a text file.
"""
import sys

def main():
    """ Demonstrate writing and reading to a text file. """
    movies = { 'Donald': ['Braveheart', 'Brave', 'Brigadoon'],
               'Mira': ['Matrix', 'Mad Max', 'Magnolia'],
               'Sarah': ['Seven', 'Scream', 'Saving Private Ryan']
    }
    filename = r"C:\labs\movies.txt" # Always use a raw string for
        paths.

    # Open file handle for WRITING in text mode.
    fh_out = open(filename, mode="wt")

    # Iterate through Movie keys and write key: objects to file
    # handle.
    for name in movies.keys():
        print(f"{name}: {movies[name]}", end="\n")
        fh_out.write(f"{name}: {movies[name]}\n") # Remember
            newline.
    fh_out.close() # Flush buffers and close file handle.

    # Open file handle for READING in text mode.
    fh_in = open(filename, mode="rt")

    # text = fh_in.read() # Read ENTIRE file into str object.
    # text = fh_in.read(30)  # Read NEXT 30 chars into str object.
    # text = fh_in.readline()  # Read NEXT line into str object.
    # print(text)

    # lines = fh_in.readlines()  # Read ENTIRE file into list
    # bject.
    # print(lines)
    # print(f"1st line = {lines[0]}")

    # Iterate through file handle and read one line at a time.
    # for name in open(filename, mode="wt"):
    for name in fh_in:
        print(name, end="")
    fh_in.close()  # Flush buffers and close file handle.
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

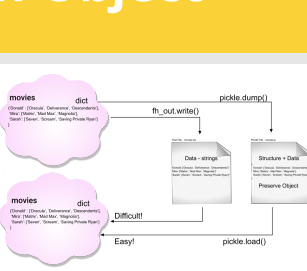
File Interaction TEXT Random Access

```
#! /bin/python
# Name:        demo_random_rw_text.py
# Author:      QA2.0, Donald Cameron
# Revision:    v1.0
# Description: This program will demonstrate how to Interact with
    the
# file system by opening/closing file handles random reading using
    the
# seek() and tell() file handle methods.
"""
    Read movie data randomly from a TEXT file.
"""
import sys

def main():
    """ Demonstrate random reading from a TEXT file. """

    filename = r"C:\labs\movies.txt" # Always use a raw string for
        paths.
    SOF = 0 # Start of file.
    CUR = 1 # Current Position (only on binary file).
    EOF = 2 # End of file (only on binary file).

    # Open file handle for READING in TEXT mode.
    fh_in = open(filename, mode="rt")

    # Seek to char position 50 from SOF and read next 30 chars.
    fh_in.seek(50, SOF)
    text = fh_in.read(30)
    print(f"Text = {text} at position {fh_in.tell() - len(text)}")

    # Seek to char position 90 from SOF and read next 40 chars.
    fh_in.seek(90, SOF)
    text = fh_in.read(40)
    print(f"Text = {text} at position {fh_in.tell() - len(text)}")

    fh_in.close()  # Flush buffers and close file handle.
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

File Interaction BINARY Random Access

```
#! /bin/python
# Name:         demo_random_rw_binary.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to Interact with
    the
# file system by opening/closing file handles random reading using
    the
# seek() and tell() file handle methods.
"""
    Read movie data randomly from a binary file.
"""
import sys

def main():
    """ Demonstrate random reading from a BINARY file. """

    filename = r"C:\labs\movies.txt" # Always use a raw string for
        paths.
    SOF = 0 # Start of file.
    CUR = 1 # Current Position.
    EOF = 2 # End of file.

    # Open file handle for READING in BINARY mode.
    fh_in = open(filename, mode="rb")

    # Seek to char position 50 from SOF and read next 30 bytes.
    fh_in.seek(50, SOF)
    text = fh_in.read(30)
    print(f"Text = {text} at position {fh_in.tell() - len(text)}")

    # Seek forwards 30 chars from current position and read next 40
    # bytes.
    fh_in.seek(30, CUR)
    text = fh_in.read(40)
    print(f"Text = {text} at position {fh_in.tell() - len(text)}")

    # Seek backwards 40 chars from EOF and read next 20 bytes.
    fh_in.seek(-40, EOF)
    text = fh_in.read(20)
    print(f"Text = {text} at position {fh_in.tell() - len(text)}")

    fh_in.close()  # Flush buffers and close file handle.
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Pickle an Object



Pickling converts any Python object, except functions, into a stream of bytes - preserving the Data plus its STRUCTURE!

Resultant pickle files can be written in a number of formats - this is called the pickle protocol and is a named parameter for the pickle dump function. Choose from:

- protocol=0 (ASCII for backwards compatability)
- protocol=1 (Binary)
- protocol=2 (Binary from Python 2.3)
- protocol=3 or pickle.DEFAULT_PROTOCOL (Binary from Python 3.0)
- protocol=4 or pickle.HIGHEST_PROTOCOL (Binary from Python 3.4)

```
#! /bin/python
# Name:         demo_pickle.py
# Author:       GA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to preserve Python objects
# (data plus structure) to a file using the pickle module.
"""
    Preserve/Pickle ONE Python object to a file.
"""
import sys
import pickle
import pprint

def main():
    """ Preserve and reload a Python object a file """
    # Create a dict to hold our favourite movies.
    movies = { 'Donald': ['Dracula', 'Deliverance', 'Descendants'],
               'Mira': ['Matrix', 'Mad Max', 'Magnolia'],
               'Sarah': ['Seven', 'Scream', 'Saving Private Ryan']
    }

    # Open file handle in binary write mode for pickling.
    fh_out = open(r'C:\labs\movies.p', "wb")

    # Dump the movies dict to the pickle file using a chosen protocol.
    # pickle.dump(movies, fh_out, protocol=0) # ASCII
    # pickle.dump(movies, fh_out, protocol=4) # Binary space efficient.
    pickle.dump(movies, fh_out, pickle.DEFAULT_PROTOCOL) # protocol=3
    # pickle.dump(movies, fh_out, pickle.HIGHEST_PROTOCOL)  # protocol=4
    fh_out.close()

    # Open file handle in binary read mode for pickling.
    fh_in = open(r'C:\labs\movies.p', "rb")

    # Reload pickled dict from file back into memory.
    films = pickle.load(fh_in)
    fh_in.close()

    # Display and compare the movies and films dict using the
    # Std Py library module pprint (pretty print)
    pprint.pprint(movies)
    print('-' * 60)
    pprint.pprint(films)
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Multiple objects on a Shelve

Benefits of Python shelve over pickle

1. Dictionary-Like Interface:
- Shelve provides a key-value storage system, similar to a dictionary, making it easy to store and retrieve individual objects by key. In contrast, pickle requires manual handling of the entire serialized object.

2. Selective Access:
- With shelve, you can retrieve specific objects from the storage file without loading the entire dataset into memory.
- Pickle loads the entire serialized object, which can be inefficient for large datasets.

3. Persistent Data Management:
- Shelve is ideal for long-term storage, acting like a mini database where you can update only parts of the data.
- Pickle works best for short-term serialization, where you need to store or transmit whole objects at once.

4. File Handling Efficiency:
- Shelve handles file operations automatically, reading and writing only the accessed parts.
- With pickle, you must manually manage file opening, writing, and closing, which can be less efficient for frequent updates.

In summary, **shelve** offers easier key-based access, efficient partial data loading, and built-in file management, making it more practical for persistent storage needs compared to **pickle**.

```
#! /bin/python
# Name:         demo_shelve.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to preserve  MULTIPLE
# Python objects to a file using the shelve module.
"""
    Preserve/Pickle ONE Python object to a file.
"""
import sys
import shelve
import pprint

def main():
    """ Preserve and reload multiple Python objects a file """
    # Create dictionaries to hold our favourite movies, tv series and music bands.
    movies = { 'Donald': ['Dracula', 'Deliverance', 'Descendants'],
               'Mira': ['Matrix', 'Mad Max', 'Magnolia'],
               'Sarah': ['Seven', 'Scream', 'Saving Private Ryan']
    }

    tv = {'Donald': ['Dad's Army', 'Dragon's Den', 'Dallas'],
          'Mira': ['MASH', 'Masterchef', 'Mythbusters'],
          'Sarah': ['Scrubs', 'Sesame Street', 'Star Trek']
    }

    music = {'Donald': ['David Bowie', 'Deacon Blue', 'Duran Duran'],
             'Mira': ['Metallica', 'Madness', 'Meatloaf'],
             'Sarah': ['S Club 7', 'Spandau Ballet', 'Spice Girls']
    }


    # Open file handle in preserving multiple objects.
    db = shelve.open(r"C:\labs\movies.db")

    # Write key: objects to shelve database.
    db['movies'] = movies
    db['tv'] = tv
    db['music'] = music

    db.close()

    # Open the Shelve file handle again.
    db = shelve.open(r"C:\labs\movies.db")

    while True:
        # Iterate through db and display itemised keys.
        for key in db:
            print(key)

        # Prompt user to choose key.
        db_key = input("Choose a key to display or q|Q=quit: ")
        if db_key.lower() == "q": break
        pprint.pprint(db[db_key])

    db.close()
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Interacting with a DB

```
#! /bin/python
# Name:         demo_sqlite.py
# Author:       GA2_0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to connect to a SQLite database
# file, create a table, insert and query rows using the SQLite3 module/API.
"""
    Create and populate a SQLite database file.
"""
import sys
import sqlite3
import os


def main():
    """ Create, Insert and query all rows in database """
    # Prompt user if they want to create or use existing database file.
    if os.path.isfile(r"C:\labs\movies_db.sqlite"):
        response = input("Do you want to use existing SQLite file (y/n)? ")
        if response.lower() == "n":
            os.remove(r"C:\labs\movies_db.sqlite")

    # Connect into SQLite database file.
    db_conn = sqlite3.connect(r"C:\labs\movies_db.sqlite")
    # Open a cursor to store
    cur = db_conn.cursor()

    # Create a movies table if it doesn't already exist.
    cur.execute("""CREATE TABLE IF NOT EXISTS movies
                    ( id      INTEGER PRIMARY KEY
                    ,name    TEXT
                    ,year    DATE
                    ,rating  INTEGER ) """)

    # Loop and write users favourite movies to database file.
    while True:
        movie = input("Enter name of movie (q=quit): ")
        if movie.lower() == "q": break
        year = input(f"Enter year of release for {movie}: ")
        rating = input(f"Enter your rating for {movie} (1-10): ")

        cur.execute("""INSERT INTO movies (name,year,rating) VALUES(?,?,?)""", (movie, year
        , rating))
        print("1 row inserted")

    db_conn.commit()  # Commit Changes to database.

    # Query, fetch and display all rows from database file.
    cur.execute("""SELECT * FROM movies""")
    rows = cur.fetchall()
    for row in rows:
        print(row)

    db_conn.close() # Close database connection.
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Quiz

Persistent Data Knowledge Check

5-10 mins

1. What is the default mode for the built-in open() function?

To be revealed

2. What is the default buffering mode for text files?

To be revealed

3. Which methods are used for random access?

To be revealed

4. Which Python standard library module preserves ONE Python object to a file?

To be revealed

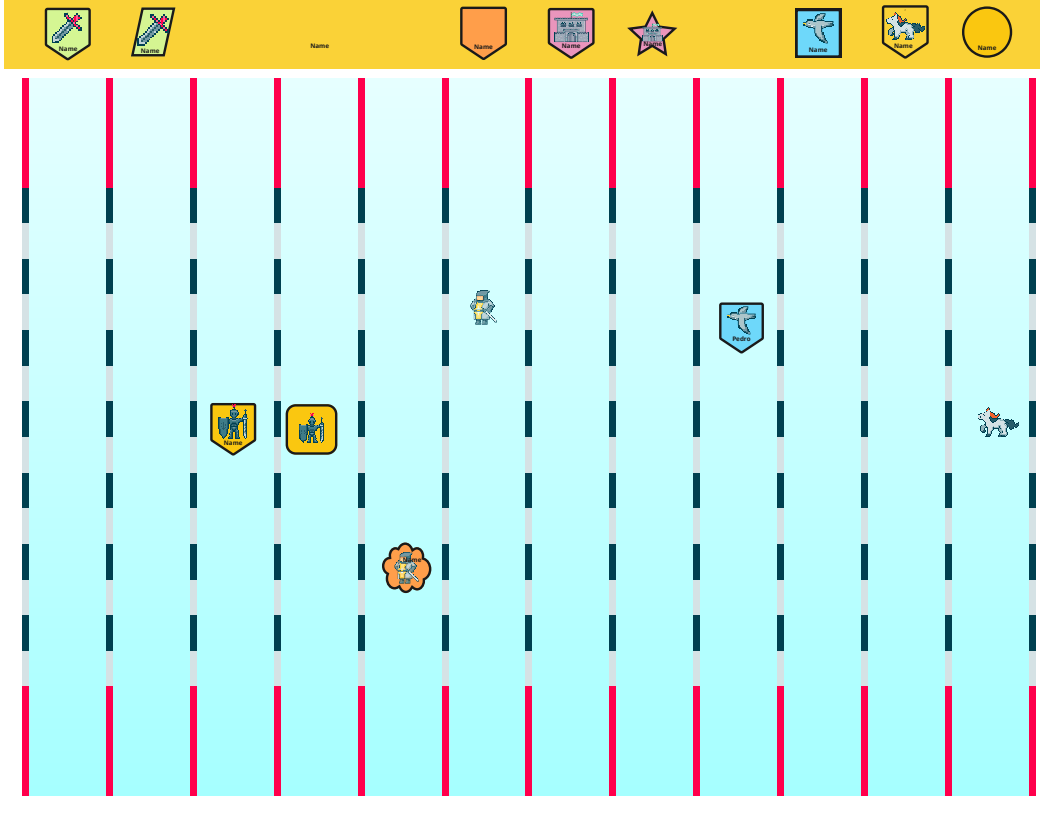
5. Which Python standard library module preserves MULTIPLE Python objects to a file?

To be revealed

Exercises

35 mins

Move your piece down as you complete the exercises



Persistent Storage – exercises for 'peasants'

1 **Open the file C:\labs\top250_movies.txt.** Notice it contains the top 250 movies as voted by Letterboxd users.

2 Create a script called **display_movies.py** in the **C:\labs** folder that will **open** and retrieve the movies one at a time. Display the movies names and remember to **close** the file when finished. Use f-strings to display the movie information.

3 You may have noticed that the movie titles are in lower case. Modify the script so that each word of the movie title is capitalised.

4 Modify the script to display the ranking from 1 to 250 next to the movie title. Use the built-in enumerate() function to generate the ranking.

Persistent Storage – stretch exercises for Kings

5 Create a new script **C:\labs\movie_data.py**. Using the top 5 movies from the previous exercise, create a LIST called **movies** which has five dictionaries with the following keys "title", "director", and "year" released. Movie information can be found at https://www.imdb.com/.

Write a loop to iterate through the list of dictionaries. Write out the formatted movie information to a file called **C:\labs\top5_movie_info.txt**. Use the print function to write instead of the write() method.

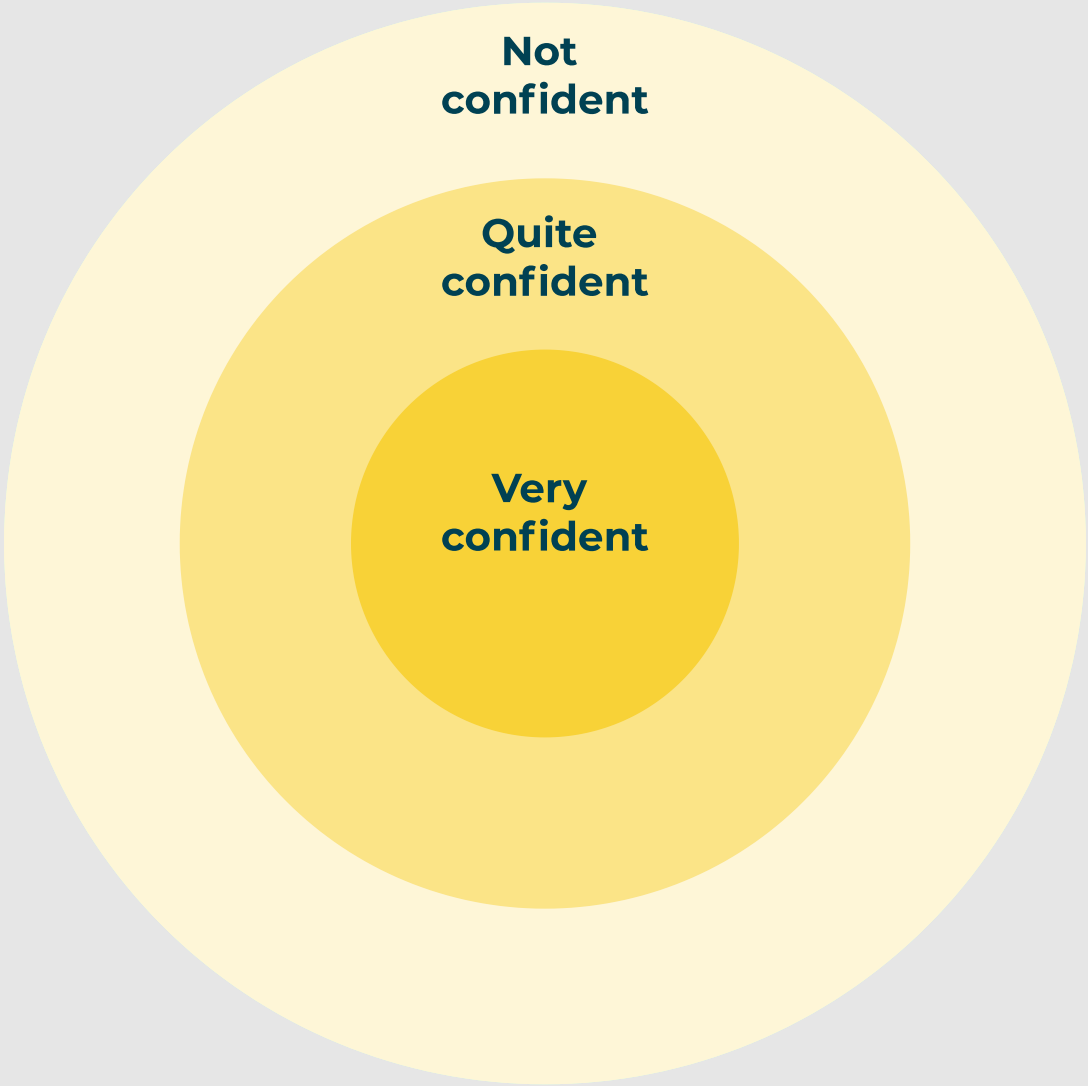
Year should be right justified 10 characters. Title should be proper-cased, and 30 characters left justified. And director right justified by 25 characters. For example:

```
1994 - Title: The Shawshank Redemption      Director:          Frank Darabont
1972 - Title: The GodFather                 Director:     Francis Ford Coppola
```

6 Think how about difficult it would be to reload this information back into a suitable Python data structure. Copy the **C:\labs\movie_data.py** file to **C:\labs\movie_dict.py** and modify the new script to **preserve** the movie list to a compressed pickle file called **C:\labs\top5movies.pgz**. Load the object back into memory and compare with the original movies dictionary [Hint: use pprint].

7 And now for something a little different.
Write a Python script called **C:\labs\display_unused_ports.py** that lists all the unused port numbers between 1 to 200 in the SERVICES file. This file is used by applications and TCP/IP to translate network service names into port numbers, for example, http and port 80. Your script should take into account which platform it is running on and select '/etc/services' on Linux and **'C:\WINDOWS\System32\drivers\etc\services'** on Windows.

Some hints to get you started:
- Check platform (i.e., are we running on Windows or Linux?) and use correct file pathname
- Write code to iterate through file and display one line at a time
- Ignore blank lines and comments
- We've not met Regular Expressions yet, so consider using string splitting to isolate the number
- Extract port numbers (will have to convert to integer)
- Port numbers can be duplicated so store numbers in a suitable object which filters duplicates



Back to the quest

match groups
assertions
pre-compile
group basic end substitution
repetition
escape expressions
regex
groupings regular extended
alternation start
objects quantifiers

Have you used Regular Expressions before?

Comment where you have used them.

<Add a Sticky Note>

Basic Regex

Regular Expressions (Regex, or just RE) are used in many tools, sql, and programming languages, and have their roots in Unix command line tools such as ed, ex, grep, sed, and awk. They provide a more extensive and powerful set of meta-characters for pattern matching and come in several dialects. The Python standard library module re.py provides matching functions and support for **Basic**, **Extended** and **Python** pattern characters.

To learn how to use Regex, we need to do two things – learn the theory behind the meta-characters, and then how to apply them using Python code.

The website https://regex101.com/ is useful for learning and testing your Python Regular Expressions.

Basic Regular Expressions B.R.E

Line Anchors

Start `^The`

End `ing$`

Single Char Class

Any char `^.ing$`

Any char `^.....$`

Single Char Range Class

Char set `^[rdz]ing$`

Char range `^[A-Z]ing$`

Char ranges `^[A-Za-z]ing$`

Char NOT set `^[^dz]ing$`

Multiple sets `[aeiou][aeiou][aeiou]`

Escape Char Class

Escape char `\.`

Repetition Class

0 or more `:[0-9]*:`

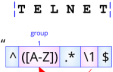
Limiting Repetition

Exact `:[0-9]{10}:`

Min, max `:[0-9]{10,20}:`

At least `:[0-9]{10,}:`

Grouping/Back Referrals



Extended RE

Extended Regular Expressions or E.R.E added some extra repetition chars (?+), groupings (rhubarb)+ and alternation (or).

Extended Regular Expressions E.R.E

Repetition Class

0 or more `:[0-9]*:`

0 or once `:[0-9]?:`

1 or more `:[0-9]+:`

Alternation Class

or `eric|graham|michael`

Grouping

`a (bottle|glass|keg) of (lager|wine|beer)`

- a bottle of lager
- a glass of wine
- a glass of beer

Python Assertions/Escape chars

Escape Chars

Digit	\d	[0-9]	\D	[^0-9]
Word char	\w	[A-Za-z0-9]	\W	[^A-Za-z0-9]
Whitespace	\s	[\t\r\f\n]	\S	[^ \t\r\f\n]
Word boundary	\b	[\t\n,.';:"-+()!?]	\B	[^ \t\n,.';:"-+()!?]
Line anchors	\A	^	\Z	$

Regex Code

There are many online resources for learning Regular Expressions with Python. Here are a few to start with:
https://regex101.com/
https://pythex.org/
https://extendsclass.com/regex-tester.html
http://www.pyregex.com/

```
#! /bin/python
# Name:         demo_regex_1.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to match string
# data using Regular Expressions.
"""
    Search a given file using regular expressions and display
    matched lines.
"""
import sys
import re

def main():
    """ Demonstrate different 4_Regular_Expressions patterns """
    fh_in = open(r"C:\labs\words", mode="rt")

    # Iterate through file handle
    for line in fh_in:
        # m = re.search(r"^the", line)  # Match lines starting with 'the'.
        # m = re.search(r"ing$", line)  # Match lines ending with 'ing'.
        # m = re.search(r"^[A-Z]", line)  # Match lines starting with a capital

        # m = re.search(r"\.", line)  # Match lines with a dot.
        # m = re.search(r"[aeiou][aeiou][aeiou]", line)  # Match 3 consecutive
            vowels.
        # m = re.search(r"[aeiou]{5,}", line)  # Match at least 5 consecutive
            vowels.
        # m = re.search(r"^.....$", line)  # Match 19 char lines.
        # m = re.search(r"^.{19}$", line)  # Match 19 char lines.
        # m = re.search(r"^(.)(.).\2\1$", line)  # Match lines 5 char
            palindromes.
        # m = re.match(r"(.)(.).\2\1$", line)  # Match 5 char palindromes using
            match().
        m = re.search(r"^(.).*\1$", line)  # Match lines starting/ending with
            same char.

        if m:
            print(line, end="")

    fh_in.close()
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Regex Groups

The functions search(), match(), fullmatch(), sub(), and subn() either return None or an 're.Match' object if a pattern is matched. The match object has several methods for returning information about the pattern match.

- m.group() # Returns the matched pattern.
- m.start() # Returns the char position where pattern starts.
- m.group() # Returns the char position where the pattern ends.
- m.groups() # Returns a tuple of groupings, or error if no groupings in pattern.
- m.groups()[0] # Returns the 1st grouping from the tuple. Index = 0, grouping = 1st. Index = 1, grouping = 2nd.
- m.group(1) # Also returns the 1st grouping and is more Pythonic. That is, parameter = 1, grouping = 1st.

```
#! /bin/python
# Name:         demo_regex_2.py
# Author:       QA2_0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to match string
# data using Regular Expressions.
"""
    Search a given file using regular expressions and display
    matched lines and groupings.
"""
import sys
import re

def main():
    """ Demonstrate 4_Regular_Expressions groupings """
    fh_in = open(r"C:\labs\words", mode="rt")

    # Iterate through file handle.
    for line in fh_in:
        # Match 5 char palindromes.
        m = re.search(r"^(.)(.).\2\1$", line) # Return None or RE Match object.
        if m:
            # The match object m has several methods with info about the match.
            print(f"Matched {m.group()} at char pos {m.start()} - {m.end()}, "
                  f"with groups {m.groups()}, "
                  f"first group is {m.groups()[0]}, " # Index tuple 0 = 1st
                  f"or {m.group(1)}") # More Pythonic way as 1 = 1st.

    fh_in.close()
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Pre-compile

Notice that in the previous demo we are compiling the pattern for every loop and line that is read from the file. So, if the file has 42k lines then the pattern will be compiled (into low level byte code) 42,000 times – even though the pattern never changes for the life of the loop and file handle.

In this case, there would be significant performance improvements in pre-compiling the pattern just once before the loop. The re.py module has a function called compile() which returns a Regex Object with the compiled pattern stored within. This object inherits the search(), match(), and fullmatch() functions.

```
#! /bin/python
# Name:        demo_regex_3.py
# Author:      QA2.0, Donald Cameron
# Revision:    v1.0
# Description: This program will demonstrate how to match
    str
# data using Regular Expressions and pre-compiling the
    pattern once.
"""
    Search a given file using regular expressions and
        display
    matched lines.
"""
import sys
import re

def main():
    """ Display lines in a given file which match Regex """
    fh_in = open(r"C:\labs\words", mode="rt")

    # Pre-compile the pattern once if it does not change.
    reobj = re.compile(r"^.{19}$")

    # Iterate through file handle
    for line in fh_in:
        # Pre-compiled pattern stored in reobj object.
        m = reobj.match(line) # Return None or RE Match
            object.
        if m:
            print(line, end="")

    fh_in.close()
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Substitution

The re.py module also provides two functions for performing substitutions within a string. The sub() function can search, make a change, and return a modified string. Whilst the subn() function returns a tuple of the modified string and the number of changes made. All the RE functions support optional flags.

Optional Flags

Long Name	Short Name	Embedded	Comment
re.IGNORECASE	re.I	(?i)	Case insensitive match
re.MULTILINE	re.M	(?m)	^ and $ match start and end of line
re.DOTALL	re.S	(?s)	Dot '.' also matches a newline "\n"
re.VERBOSE	re.X	(?x)	Whitespace is ignored within pattern - to allow embedded comments.

```
#! /bin/python
# Name:         demo_regex_4.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to match and substitute string
# data using the re.py module and the sub() and subn() functions. It will also
# introduce Regex flags such as ignore case.
"""
    Search and substitute strings using regular expressions.
"""
import sys
import re

def main():
    """ Executed directly when run as a program """
    # This str object is storing login data for the root user from a
    # a file /etc/passwd on the Linux file system.
    line = "root:x:0:0:The root User:/root:/bin/sh"

    # The sub() function returns the modified string.
    line = re.sub(r"[rR]oot [Uu]ser", r"Administrator", line)
    print(f"Modified line = {line}")

    # The sub() function returns the modified string.
    line = re.sub(r"/bin/sh", r"/bin/bash", line)
    print(f"Modified line = {line}")

    # 4_Regular_Expressions are GREEDY and match the largest pattern by default
        .
    line = "root:x:0:0:The root User:/root:/bin/sh"
    line = re.sub(r"^(root).*\1", r"I am greedy", line)
    print(f"Modified line = {line}")

    # The subn() function returns a TUPLE of (modfiied string, num changes).
    line = "root:x:0:0:The root User:/root:/bin/sh"
    (line, num) = re.subn(r"[rR]oot", r"Groot", line)
    print(f"Modified line = {line} with {num} changes")

    # Search and replace string with optional RE flags (introduced Py3.1).
    line = "root:x:0:0:The root User:/root:/bin/sh"
    # The optional flags can be set several different ways.
    line = re.sub(r"root user", r"Administrator", line,
                  flags=re.IGNORECASE|re.MULTILINE)
    line = re.sub(r"root user", r"Administrator", line, flags=re.I|re.M)
    line = re.sub(r"(?im)root user", r"Administrator", line)
    line = re.sub(r"(?i:r)oot (?i:u)ser", r"Administrator", line) # Python 3.6.
    print(f"Modified line = {line}")

    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Quiz

Regular Expression Knowledge Check

5-10 mins

1. **What does B.R.E and E.R.E stand for?**

To be revealed

2. **What prefix should you use with your Regex patterns?**

To be revealed

3. **Can you think of a better way to write the following pattern, which is 15 chars long?**

r"^.....$"

To be revealed

4. **Given this code, what will the pattern match?**

```
>>> text = "Brave Sir Robin ran away. Bravely ran away away. When danger
reared it's ugly head, he bravely turned his tail and fled."
>>> m = re.search(r"(brave).*\1", text, flags=re.IGNORECASE)
>>> print(m.group())
```

To be revealed

5. **Given this code, what would be printed?**

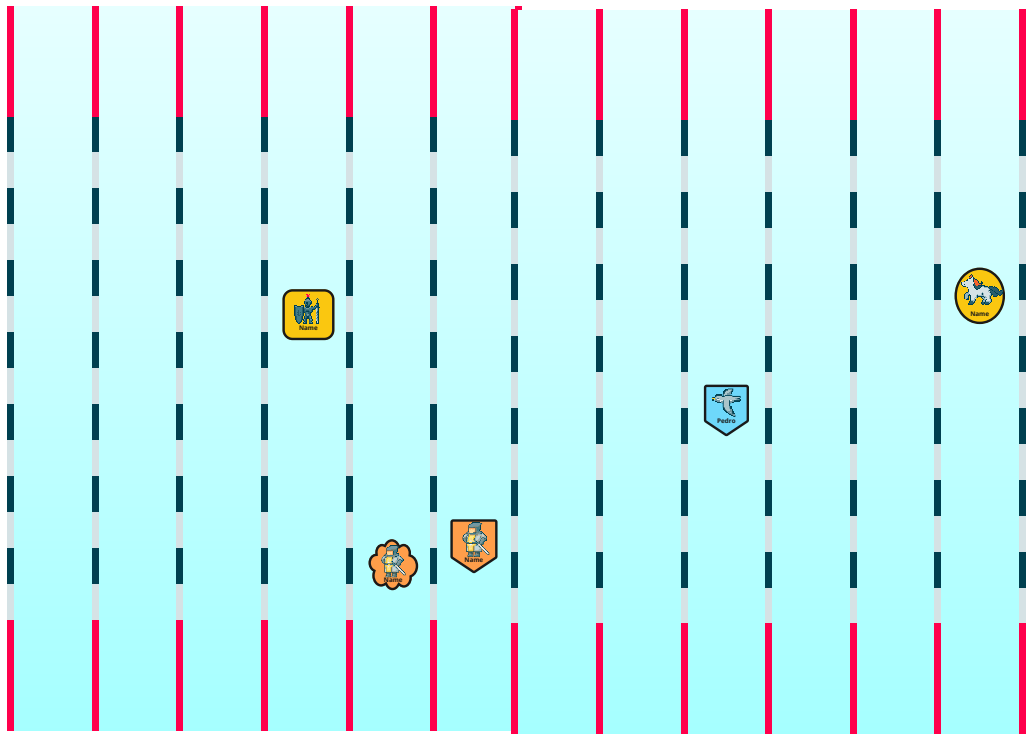
```
>>> text = "racecar"
>>> m = re.search(r"^(.)(.)(.).\3\2\1$", text)
>>> print(m.groups(), m.groups()[0], m.group(1))
```

To be revealed

Exercises

40 mins

Move your piece down as you complete the exercises



Regular Expressions – exercises for 'brave' Knights

1

For the following exercises, we will web scrape movie data from the Letterboxd (https://letterboxd.com/) movie website. Web scraping is not covered in the LIVE phase so we have written the code to help you but we need to install the BeautifulSoup and requests packages first.

Install the package on the command line or within Pycharm, or ask your instructor for help:

- venv> pip install beautifulsoup4
- venv> pip install requests

2

Load the script **C:\labs\top250.py** into Pycharm. This script scrapes the Letterboxd website for the top 250 movies and loads into a List. Compile and run the script and confirm the output displays the top 250 movies. If the web scrape doesn't work (perhaps the webpage has changed) then read the data from the c:\labs\top250_movies.txt file.

3

Modify the script so that the movie names are displayed alongside their ranking from 1 to 250. Ensure the ranking is right justified to 4 characters so that the ':' is aligned. For example:

```
1: The Shawshank Redemption
   2: The GodFather
..
 99: North by Northwest
```

4

Using regular expressions, modify the script to prompt the user to enter their favourite movie and display its ranking, if found. Ensure that the case is ignored.

Regular Expressions – stretch exercises for Kings

5

Copy the top250.py script to a new file called search250.py in the C:\labs folder.

6

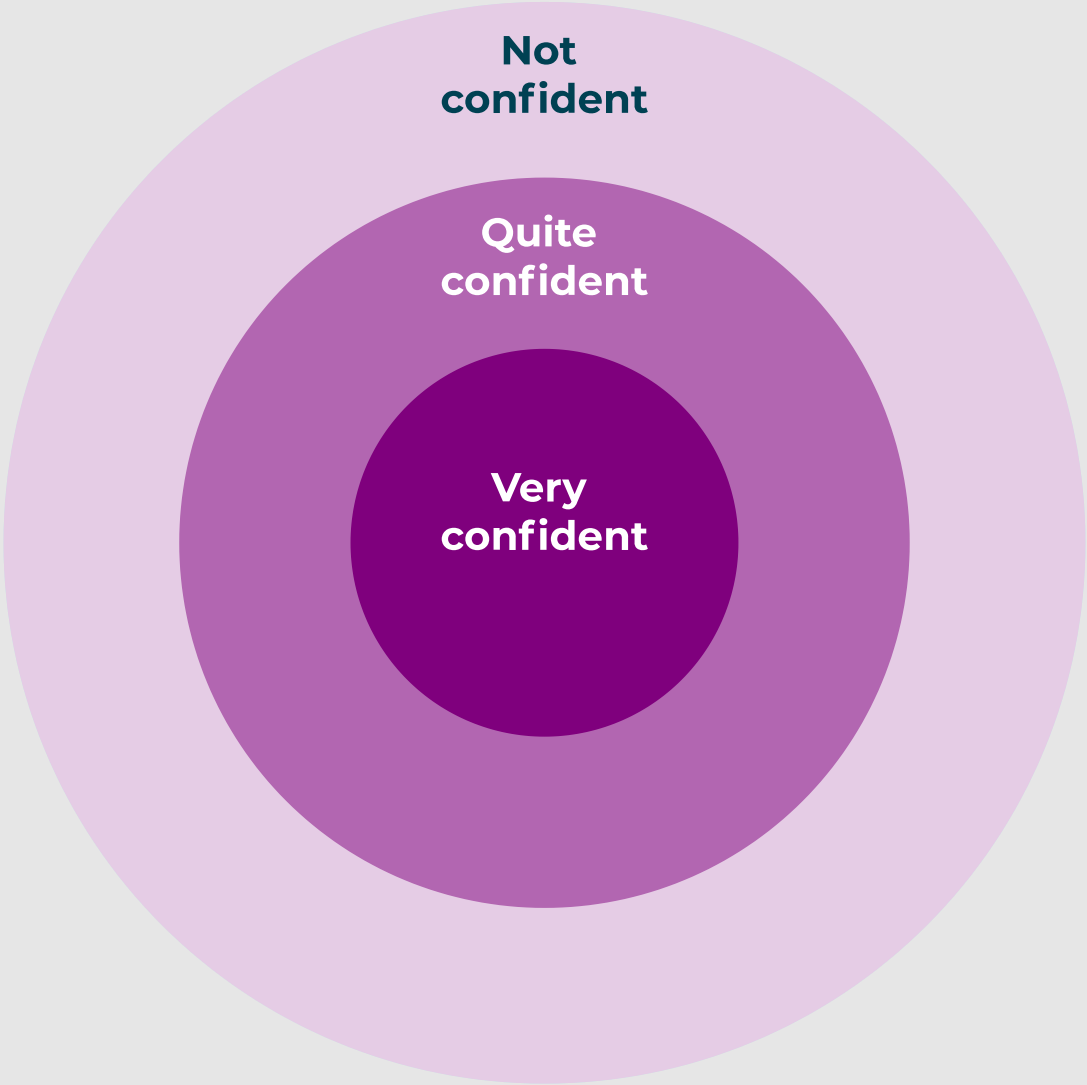
Modify the search250.py script so that it display the movies that match the following patterns and make a note of how many movies were matched for each. Ignore case for all matches.

- Movie names starting with 'the' (50)
- Movies name with a digit in the name (11)
- Movie names with 3 consecutive vowels (5)
- Movie names starting and ending in same character (17)
- Movie names only containing digits (1)
- Movies names with a repeated word in the title (4). Tricky!

The number of movies matched will change as the LIVE data from the website changes.

7

Remember the 'C:\labs\display_unused_ports.py' program from the previous lab which displays unused ports between 1 and 200? Modify the code to use Regular Expressions rather than string splitting to isolate and extract the port number.



Back to the quest

Functions

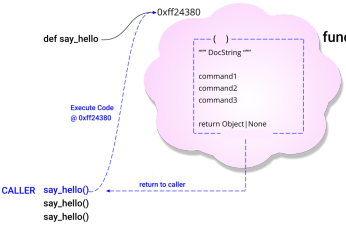
The art of reusing named blocks of code

CODE REUSE

Functions

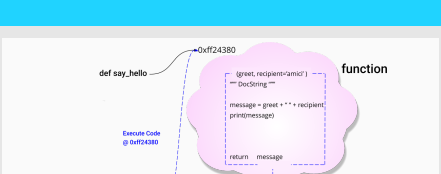
Earlier in the Digital session, you learned about functions and how useful they are for making code sharable and reusable. You saw function definitions, passing parameters to a function, returning a value from a function, and documenting functions with docstrings.

You can do more with functions! You will learn about different types of parameters, variable scope, the idea of annotating functions, lambda functions, and using functions as objects. This includes passing a function as a parameter and returning a function from another function.




```
#! /bin/python
# Name:         demo_user_functions.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to
# define, name and execute (reuse) user defined functions.
"""
    Displays greeting messages.
"""
import sys

# Example of a user defined function
def say_hello():
    """ Display the SAME GREETING to the SAME RECIPIENTS """
    message = "Hello" + " " + "World"
    print(message)
    return None

def main():
    """ Main function """
    say_hello() # CALLER - pass control to fn say_hello().
    say_hello() # Can reuse function.
    say_hello()
    say_hello()
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

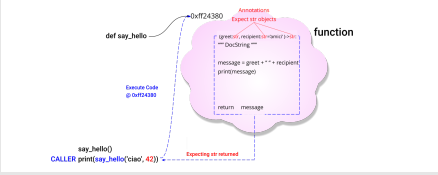
Parameters



User functions can optionally accept a finite number of parameters.
· Formal parameter names are defined within the parentheses of function declaration.
· Actual values are passed in the CALLER's parentheses.
· Literal values are passed by COPY/VALUE, e.g., (100, 200, 'hello').
· Named objects are passed by Reference, e.g., (pattern, filename). So changing a parameter within a function will alter the caller's object. To prevent this for lists and dictionaries, pass a slice of the structure, e.g., function_call(my_list[:]). It's a hack!
· Parameters can be assigned defaults, but following parameters must also have defaults.
· Parameters can be passed as positional, named, or a mix (positional followed by named).

```
#! /bin/python
# Name:         demo_user_functions_with_parameters.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to
# define, name, call and pass parameters in and return objects back.
"""
    Displays greeting messages.
"""
import sys

# Example of a user defined function with parameter passing
# and optional defaults
def say_hello(greeting="ciao", recipient="amici"):
    """ Display a given greeting to a recipients """

    # Converted parameters to strings for safety!
    message = str(greeting) + " " + str(recipient)
    print(message)

    # return None
    return message


def main():
    """ Main function - say hello to our friends """

    say_hello("Hello", "world")  # Positional parameter passing.
    say_hello(greeting="Salutare", recipient="prietenі")  # Named parameter passing.
    say_hello("Namaste", recipient="mere dost")  # Mixed parameter passing.
    say_hello(recipient="mes amis", greeting="Bonjour")  # Mixed in different order.
    say_hello("Nazdar", 42) # Try passing in an Integer!
    say_hello()

    # If the function returns an object we can then have the CALLER as
    # a r-value. That means on the RHS of an assignment or embedded within
    # a larger expression.
    # num = say_hello()
    print(f"Default message is {say_hello()}")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Annotations



Function annotations were introduced in PEP 3107. They are used to provide additional information about the function, including what data types are expected for the input parameters and return value. They are like an embedded comment, but are more than a comment (which are ignored by the compiler). They are preserved in the function attribute __annotations__.

Annotations have no meaning except 'commenting in-code' of what types of data are hoped for. But 3rd party modules might use them for additional docstrings or for enforcing data type on parameters or return values. Python will remain a dynamically typed language!

```
#! /bin/python
# Name:         demo_user_functions_annotations.py
# Author:       GA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to
# define, name, call and pass parameters in and return objects back.
"""
    Displays greeting messages.
"""
import sys

# Example of a user function ANNOTATIONS (Py2.5, not enforced)
def say_hello(greeting:str='ciao', recipient:str='amici')->None:
    """ Display a given greeting to a recipients """

    # Annotations not enforced by Python so convert parameters
    # into strings for safety.
    message = str(greeting) + " " + str(recipient)
    print(message)

    # return None
    return message


def main():
    """ Main function - say hello to our friends """

    say_hello("Hello", "world")  # Positional parameter passing.
    say_hello(greeting="Salutare", recipient="prieteni")  # Named parameter passing.
    say_hello("Namaste", recipient="mere dost")  # Mixed parameter passing.
    say_hello(recipient="mes amis", greeting="Bonjour")  # Mixed in different order.
    say_hello("Nazdar", 42) # Try passing in an Integer!
    say_hello()

    # If the function returns an object we can then have the CALLER as
    # a r-value. That means on the RHS of an assignment or embedded within
    # a larger expression.
    # num = say_hello()
    print(f"Default message is {say_hello()}")

    # Attributes of functions can be displayed including annotations.
    print(f"Annotations for say_hello() = {say_hello.__annotations__}")
    return None


if __name__ == "__main__":
    main()
    sys.exit(0)
```

Variadic Functions

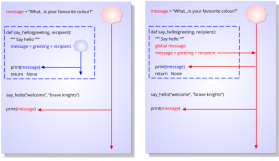
A **Variadic function** can accept a variable number of parameters. For example, the built-in function print() can accept any number of parameters. User functions can also accept variable number of parameters into either a **tuple** with a '*****' prefix or into a **dictionary** with a '******' prefix. This works in a similar way to unpacking as discussed with collections.

DEFINITION: def display_vat(**kwargs)
CALLER: display_vat(vatpc=15, gross=9.55, message='Summary') # Use named parameters when passing to a dict.

```
#! /bin/python
# Name:         demo_user_functions_variadic.py
# Author:       OA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to
# define, name, call and pass parameters in and return None.
"""
    Displays greeting messages to ALL friends and foes.
"""
import sys

# Example of a Variadic function.
def say_hello(greeting="ciao", *recipients):
    """
    Accept a greeting string and variable number of recipients and
    display message to ALL recipients and returns None.
    """

    # Use an iterator for loop to iterate through objects in tuple.
    for person in recipients:
        message = str(greeting) + " " + str(person)
        print(message)

    # return None
    return None


def main():
    """ Main function - say hello to our friends """

    # Pass in a variable number of parameters to the function.
    say_hello('None shall pass', 'green knight')
    say_hello('None shall pass', 'green knight', 'arthur', 'patsy')

    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Scope

Local Vs Global



Scope defines the **life** and **visibility** of a variable/object, in other words, what part of the code can see and use it. Python supports global scope and function/local scope. Global variables/objects are visible to the entire program including nested functions.

But if a value is assigned in the function before it is used, then it becomes a local variable (visible and lives for duration of function). To change the value of an outer global variable within a function (frowned upon), declare the variable as **global**.

```
#! /bin/python
# Name:         demo_user_functions_scope.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate the scope (life + visibility)
# of variables/objects
"""
    Displays greeting local and global messages.
"""
import sys

message = "What...is your favourite colour?"

# Example of a user defined function with parameter passing
# and optional defaults
def say_hello(greeting="welcome", recipient="brave knights"):
    """ Display a given greeting to a recipients """
    # global message # Try executing with and without this commented.

    # An assignment to message forces Python to create a new local object.
    message = str(greeting) + " " + str(recipient)
    print(message)

    # return None
    return message

def main():
    """ Main function - say hello to our friends """
    say_hello()
    print(f"function return is {say_hello()}")
    print(message)

    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Docstrings

HELP, I'm lost!

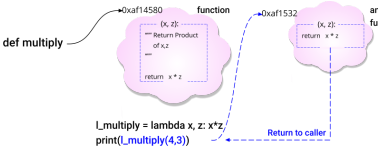
So have you been wondering why we have been putting tripled quoted strings in our programs and functions? Well, these are called DocStrings and usually appear immediately after the definition of a program/module, class, function, or method.

Docstrings can be simple one-liners or multi-line and should not be descriptive (that is for # comments). Rather they must follow "Do this, return that" format. Multi-line docstrings can be more elaborate, and are described in PEP 257, and can include details about input parameters and return values. Docstrings for classes (later in LIVE) should summarise its behaviour and list public methods. The __doc__ attribute can be used to access docstrings.

And they are read by the Python help() function – so now you know where we are getting help from for our modules and functions!

```
#! /bin/python
# Name:         demo_user_functions_docstrings.py
# Author:       GA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate another example of
# creating functions with DocStrings, parameter passing, return values.
"""
    Calculator program with Add, Divide and Multiply functionality.
"""
import sys

def multiply(x, z):
    """
    Accepts two numeric parameters and return product.

    :param x (int|float): Integer or float
    :param z (int|float): Integer or float
    :return (int|float): Numeric product of x and z
    """
    return x * z

def add(x, z):
    """
    Accepts two numeric parameters, adds them and returns sum.

    :param x (int|float): Integer or float
    :param z (int|float): Integer or float
    :return (int|float): Numeric of x summed with z
    """
    return x + z

def divide(x, z):
    """
    Accepts two numeric parameters, divides first by the second and returns result.

    :param x (int|float): Integer or float
    :param z (int|float): Integer or float
    :return (str): f-string of x divided by z to 3 decimal places.
    """
    return f"{x/z:.3f}"

def main():
    """ Single Line docstrings are ideal if the function is self explanatory """
    print(f"4 * 3 = {multiply(4, 3)}")
    print(f"4 + 3 = {add(4, 3)}")
    print(f"4 / 3 = {divide(4, 3)}")

    # An example of a docstring for the built-in print function.
    # print(print.__doc__)
    # An example of a docstring for the sys module.
    # print(sys.__doc__)
    # Displaying the docstring for user functions.
    print(multiply.__doc__)
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

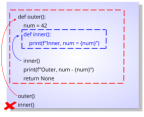
Lambda Functions




```
#! /bin/python
# Name:         demo_user_functions_lambda.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate another example of
# creating functions using lambda (anonymous) functions.
# Simple one-line docstrings are used.
"""
    Calculator program with Add, Divide and Multiply functionality.
"""

import sys

def multiply(x, z):
    """ Multiply two numeric parameters and return result """
    return x * z

def add(x, z):
    """ Add two numeric parameters and return result """
    return x + z

def divide(x, z):
    """ Divide two numeric parameters and return formatted str """
    return f"{x/z:.3f}"

def main():
    """ This is the main program function """
    print(f"4 * 3 = {multiply(4, 3)}")
    print(f"4 + 3 = {add(4, 3)}")
    print(f"4 / 3 = {divide(4, 3)}")

    # A lambda function is a small anonymous function that can take
    # multiple arguments but only has one expression. And is an
        alternative
    # way to defining simple functions without the 'def' statement.
    l_mul = lambda x, z: x * z
    l_add = lambda x, z: x + z
    l_div = lambda x, z: f"{x/z:.3f}"

    print("-" * 20)

    print(f"4 * 3 = {l_mul(4, 3)}")
    print(f"4 + 3 = {l_add(4, 3)}")
    print(f"4 / 3 = {l_div(4, 3)}")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Nested Functions

Nested function



Variables in nested functions

Python allows the nesting of functions inside each other, which not all languages support. It can be useful for simple reusable, recursive functions, function factories (a function can be returned from a function like any object) or for closures. Like nested objects, functions follow the same scope rules. Nested functions can only be called within the function they are defined.

Python has a clean, safe, and dynamic way of managing variables/objects. As soon as you make an assignment, a new object is created - either a **GLOBAL** or a **LOCAL** (within a function). This mostly works well, but on occasion we might have nested functions, and we don't quite want to create a new variable and don't want to access the global variable either. This is where the **nonlocal** keyword proves useful to indicate the variable is referring to an enclosing (outer) scope variable.

```
#! /bin/python
# Name:         demo_user_functions_nonlocal.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate that Python supports
    nested
# functions global, local and not-so local variables.
"""
    Nested functions and scope.
"""
import sys

knight = "King Arthur"

def who_goes_next():
    """ Who is next to walk the bridge of death? """
    knight = "Sir Robin" # Really?
    def brave_knights():
        nonlocal knight
        knight = "Sir Lancelot" # Of course, Lancelot is far braver!
        return None
    brave_knights()
    print(f"{knight} will go next")
    return None

def main():
    """ Who goes there? """
    who_goes_next()
    print(f"{knight} will go next")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Quiz

User Functions Knowledge Check

5-10 mins

1. What does (*,) at the beginning of the parameter list do?

To be revealed

2. What is the output of the following function call?

```
>>> def my_func(num):
>>>     return num + 10
>>>
>>> my_func(10)
>>> print(num)
```

To be revealed

3. True of False: Python functions always return a value

To be revealed

4. What is the output of the following function call?

```
>> def display_info(**kwargs):
>>>     for z in kwargs:
>>>         print(z)

>>> display_info(movie="braveheart", year="1995")
```

To be revealed

Exercises

35 mins

Move your piece down as you complete the exercises

Functions – exercises for 'brave' Knights

1

Write your own calculator program called '**C:\lab\calc.py**' with add, divide, multiply, subtract and modulus functions.

Test your functions by passing in the following parameters.

```
add(11, 5)
  divide(11,5)
  multiply(11, 5)
  subtract(11, 5)
  modulus(11, 5)
```

2

You did add Docstrings to your functions? If not, add appropriate Docstrings to each function. To test, start IDLE and load your script (File/Open), then run (<F5>). Then in Python Shell type:

```
>>> help(add)
  >>> help(multiply)
```

3

Modify the **calc.py** script to allow the add and multiply functions to accept a variable number of parameters. Test the modified functions by passing in the following parameters.

```
add(11, 5, 33, 15, 6.5)
  multiply(11, 5, 33, 15, 6.5)
```

Functions – stretch exercises for Kings

4

In the following exercises, you will use existing demonstrations and code, and improve by creating and using reusable functions.

Open the 'C:\labs\search250.py' script you created in the Regex section. Modify the script so that it has a reusable function called search_movie() which accepts one parameter - the Regex pattern to search. Test the function by calling it from the main() function with the same patterns from the previous lab exercises.

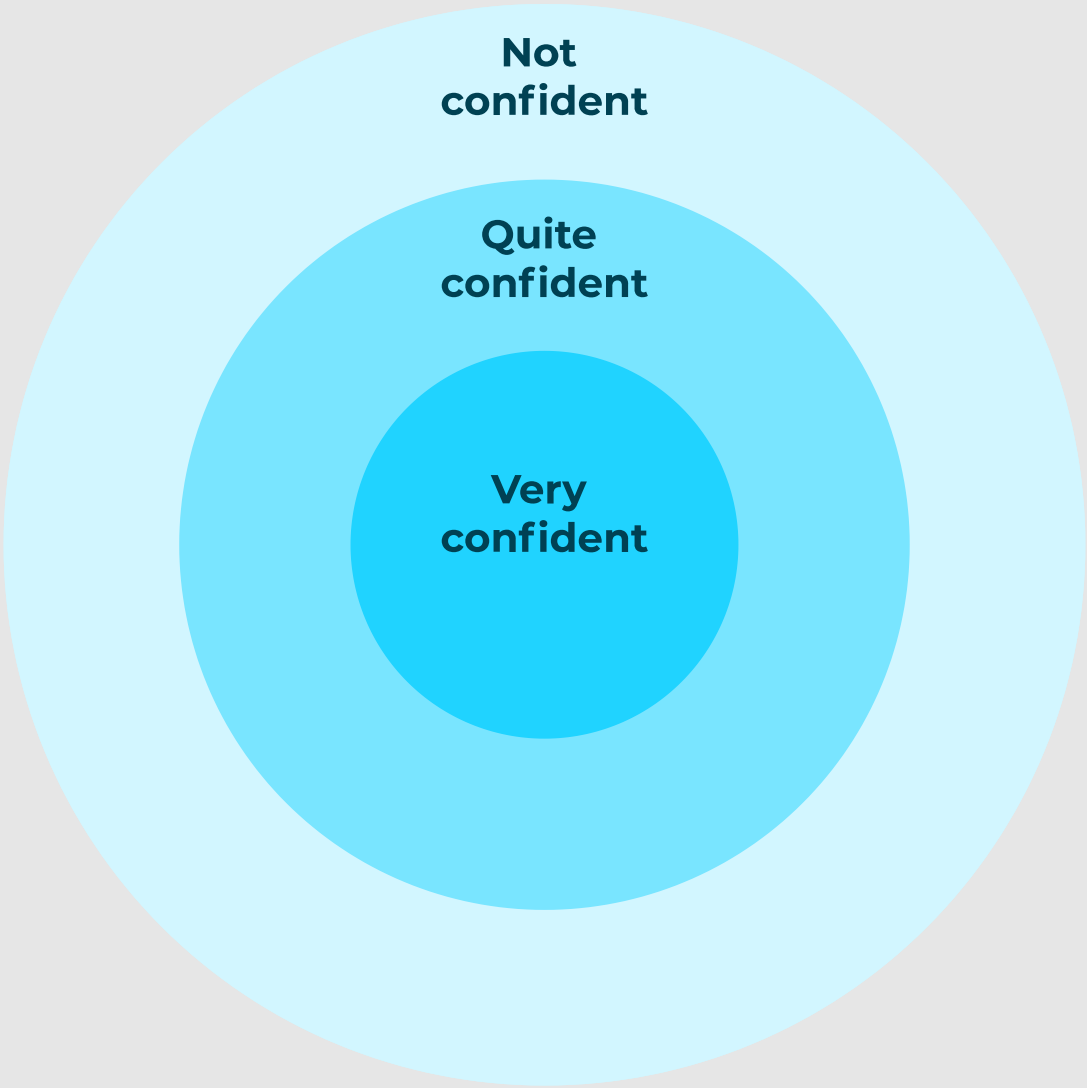
5

Open the 'C:\labs\searchWords.py' script. It searches for regular expressions patterns in the file 'C:\labs\words' and displays them to the Python shell. Modify the script and create a variadic function called search_pattern() which can accept one regex pattern followed by one or more files.

a. If no files are given, then default to 'C:\labs\words'.
b. Iterate through each file printing out lines that match.

Test with the following function calls in the main() function:-

```
search_pattern(r"^([A-Z]).*\1$")
  search_pattern(r"^([A-Z]).*\1$", r"C:\labs\words",  r"C:\labs\words")
```

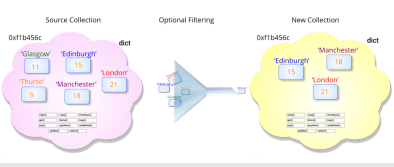


Back to the quest

Advanced Collections

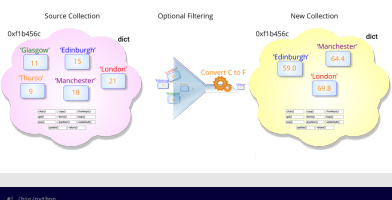
Interesting and cool things you can do with collections

FILTERING



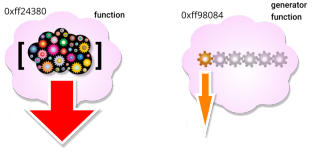

```
#! /bin/python
# Name:         demo_collections_comprehensions.py
# Author:       GA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to filter collections using
# the built-in filter() function, lambda functions and comprehensions.
"""
    Filter collections of cites into warm cities in Celsius.
"""
import sys

weather = {'Glasgow': 11,
           'London': 21,
           'Edinburgh': 15,
           'Manchester': 18,
           'Thurso': 9
           }

def filter_city(city):
    """ Filter cities by temp """
    if weather[city] >= 15:
        return True
    else:
        return False


def main():
    """ Filter collection using filter(), lambda and comprhensions """
    # For Loop plus source collection, optional if condition, and expression.
    warm_cities = {}
    for city in weather:            # Iterator for loop + collection.
        if weather[city] >= 15:         # Optional condition (filtering)
            warm_cities[city] = weather[city]  # Expression
    print(f"1.Warm Cities = {warm_cities}")

    # For Loop plus source collection, optional if condition with user function, and expression.
    warm_cities = {}
    for city in weather:
        if filter_city(city):
            warm_cities[city] = weather[city]
    print(f"2.Warm Cities = {warm_cities}")

    # Built-in filter() function plus source collection, user function for filtering.
    warm_cities = list(filter(filter_city, warm_cities))
    print(f"3.Warm Cities = {warm_cities}")

    # Built-in filter() function plus source collection, lambda function for filtering.
    wee_names = list(filter(lambda city: weather[city] >= 15, warm_cities))
    print(f"4.Warm Cities = {wee_names}")

    # For Loop plus source collection, optional if condition, and expression. LIST comprehension.
    warm_cities = [ city for city in weather if weather[city] >= 15 ]
    print(f"5.Warm Cities = {warm_cities}")

    # For Loop plus source collection, optional if condition, and expression. LIST comprehension.
    warm_cities = [ (city, weather[city]) for city in weather if weather[city] >= 15 ]
    print(f"5.1.Warm Cities = {warm_cities}")

    # For Loop plus source collection, optional if condition, and expression. DICT comprehension.
    warm_cities = { city: weather[city] for city in weather if weather[city] >= 15 }
    print(f"5.2.Warm Cities = {warm_cities}")

    # For Loop plus source collection, optional if condition, and expression. SET comprehension.
    warm_cities = { city for city in weather if weather[city] >= 15 }
    print(f"5.3.Warm Cities = {warm_cities}")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

FILTERING +



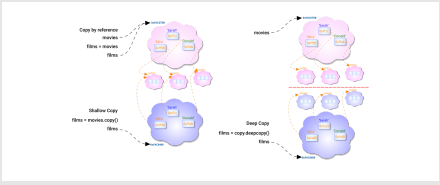

```python
#! /bin/python
# Name:         demo_collections_comprehensions_plus_c2f.py
# Author:       GX2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to filter collections using
# the built-in filter() function, lambda functions and comprehensions.
"""
    Filter collections of cities into warm cities in fahrenheit.
"""
import sys

weather = { 'Glasgow' : 11,
            'London' : 21,
            'Edinburgh' : 15,
            'Manchester' : 18,
            'Thurso' : 9
            }

def filter_city(city):
    """ Filter cities by temp """
    if weather[city] >= 15:
        return True
    else:
        return False

def c2f(temp):
    """ Accept temp in celsius and return temp in fahrenheit"""
    return (temp * 9.5) + 32

def main():
    """ Filter collection using filter(), lambda and comprhensions """
    # For loop plus source collection, optional if condition, and expression.
    warm_cities = {}
    for city in weather: # Iterator for loop + collection.
        if weather[city] >= 15: # Optional condition (filtering)
            warm_cities[city] = c2f(weather[city])  # c2f Expression
    print(f"1.Warm Cities = {warm_cities}")

    # For loop plus source collection, optional if condition with user function, and expression.
    warm_cities = {}
    for city in weather:
        if filter_city(city):
            warm_cities[city] = c2f(weather[city])
    print(f"2.Warm Cities = {warm_cities}")

    # Built-in filter() function plus source collection, user function for filtering.
    warm_cities = list(filter(filter_city, warm_cities))
    print(f"3.Warm Cities = {warm_cities}")

    # Built-in filter() function plus source collection, lambda function for filtering.
    warm_cities = list(filter(lambda city: weather[city] >= 15, warm_cities))
    print(f"4.Warm Cities = {warm_cities}")

    # For Loop plus source collection, optional if condition, and expression. LIST comprehension.
    warm_cities = [ c2f(weather[city]) for city in weather if weather[city] >= 15 ]
    print(f"5.Warm Cities = {warm_cities}")

    # For Loop plus source collection, optional if condition, and expression. LIST comprehension.
    warm_cities = [ (city, c2f(weather[city])) for city in weather if weather[city] >= 15 ]
    print(f"5.1.Warm Cities = {warm_cities}")

    # For Loop plus source collection, optional if condition, and expression. DICT comprehension.
    warm_cities = { city: c2f(weather[city]) for city in weather if weather[city] >= 15 }
    print(f"5.2.Warm Cities = {warm_cities}")

    # For Loop plus source collection, optional if condition, and expression. SET comprehension.
    warm_cities = { city for city in weather if weather[city] >= 15 }
    print(f"5.3.Warm Cities = {warm_cities}")

    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

GENERATING




```
#! /bin/python
# Name:         demo_collections_generator_functions.py
# Author:       GA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to generate a collection of
# numbers using generator functions.
"""
    Generate and display collection of numbers lazily...
"""
import sys

def get_numbers():
    """ Return an ENTIRE list of numbers """
    print("Executing get_numbers()...")
    numbers = []
    for x in range(0, 10):
        numbers.append(x)
    return numbers

def generate_numbers():
    """ GENERATE one object/number at a time = Lazy List """
    print("Executing generate numbers()...")
    for x in range(0, 10):
        yield x

def generate_numbers2():
    """ GENERATE one object/number at a time = Lazy List
        using a return and list comprehension, eek! """
    print("Executing generate numbers2()...")
    return ( x for x in range(0, 10) if x%2==0 )


def main():
    """ Generate colleciton of numbers """

    # Try each of these with large numbers for range().
    for z in get_numbers():
        print(z)

    for z in generate_numbers():
        print(z)

    for z in generate_numbers2():
        print(z)

    # Alternative ways to use a generator function and access a lazy
    # list using the builtin next() function and combined with a while loop.
    gen = generate_numbers()
    while True:
        num = next(gen, -1)
        if num != -1:
            print(num)
        else:
            break

    # Alternatively, manually get the next() object...
    gen = generate_numbers()
    num1 = next(gen, False)
    num2 = next(gen, False)
    num3 = next(gen, False)

    print(num1, num2, num3)
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

COPYING




```python
#! /bin/python
# Name:         demo_collections_copying.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to copy
    collections
# by memory reference, shallow copy and deep copy.
"""
    Copying 2_Collections.
"""
import sys
import pprint
import copy

def main():
    """ Copy 2_Collections """
    movies = { 'Donald': ['Dracula', 'Deliverance', 'Descendants'],
               'Mira': ['Matrix', 'Mad Max', 'Magnolia'],
               'Sarah': ['Seven', 'Scream', 'Saving Private Ryan']
    }

    # Comment each of the following on at a time!
    films = movies # Copy by memory reference
    # films = movies.copy() # Shallow Copy, nested object shared
    # films = copy.deepcopy(movies) # Deep copy to all levels.

    movies['Mira'][1] = 'Magnificent Seven'
    movies['Brian'] = ['Braveheart', 'Brave', 'Babe']

    pprint.pprint(movies)
    print("-" * 60)
    pprint.pprint(films)
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Quiz

Advanced Collections Knowledge Check

5-10 mins

1. **What type of comprehension is this, and what would it return?**

```
>>> [ num for num in range(0, 10) if num % 2 == 0 and num % 4 == 0]
```

To be revealed

2. **How many Gumbies (might have to google that!) would be displayed?**

```
>>> gumbys = ['eric', 'michael', 'terry', 'john', 'terry', 'graham']
>>> { name for name in gumbys if len(name) <= 5 }
```

To be revealed

3. **Would this sequence of commands return True or False?**

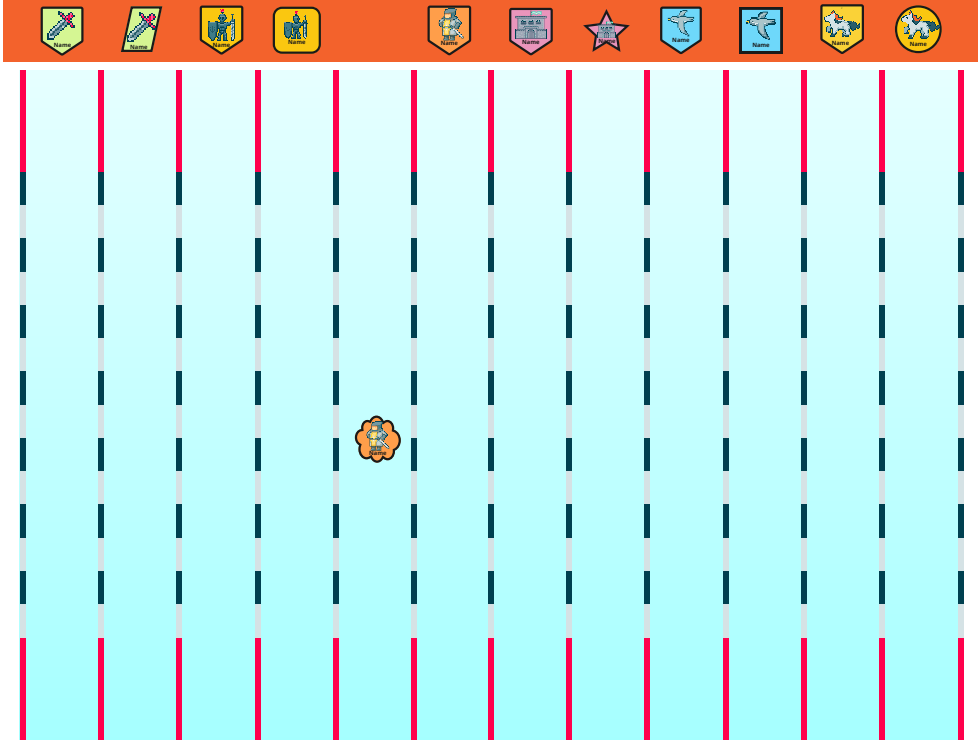
```
>>> movies = {'Cruella': '2021', 'Mulan': '2020', 'Mary Poppins Returns': '2018'}
>>> films = movies.copy()
>>> movies['Mulan'] = 1998
>>> films == movies
```

To be revealed

Exercises

35 mins

Move your piece down as you complete the exercises



Advanced Collections – exercises for 'brave' Knights

1

You should be familiar with the built-in range() function that generates a sequence of integers from a start point to a stop point with an optional step increment.

```
range(stop) # default start=0, step=1
  range(start, stop[, step])
```

Unfortunately, it only works with integers. In this exercise, we will create our own simple version of the built-in range() function but for floating point numbers. Create a new script 'C:\labs\gen.py' with a generator function called frange() which accepts at least two parameters (start, stop) and an optional parameter with default (step=0.25)). Be wary of the possibility of a step = zero being passed in!

```
  frange(start, stop[, step=0.25])

Test with the following calls in main():
  print(list(frange(1.1, 3)))
  print(list(frange(1, 3, 0.33)))
  print(list(frange(1, 3, 1)))  # Should print [1.0, 2.0].
  print(list(frange(3, 1)))  # Should print an empty list.
  print(list(frange(1, 3, 0)))  # Should print an empty list.
  print(list(frange(-1, -0.5, 0.1)))
  print(frange(1, 2))    # Should print <generator object frange at 0x...>

  for num in frange(3.142, 12):
```

2

Modify your 'c:\labs\gen.py' script and enhance your frange() function so that it can accept only one parameter that acts as the stop of the sequence. The starting value for the range should then default to 0, with step set to 0.25.

Test the function with the following code in main():

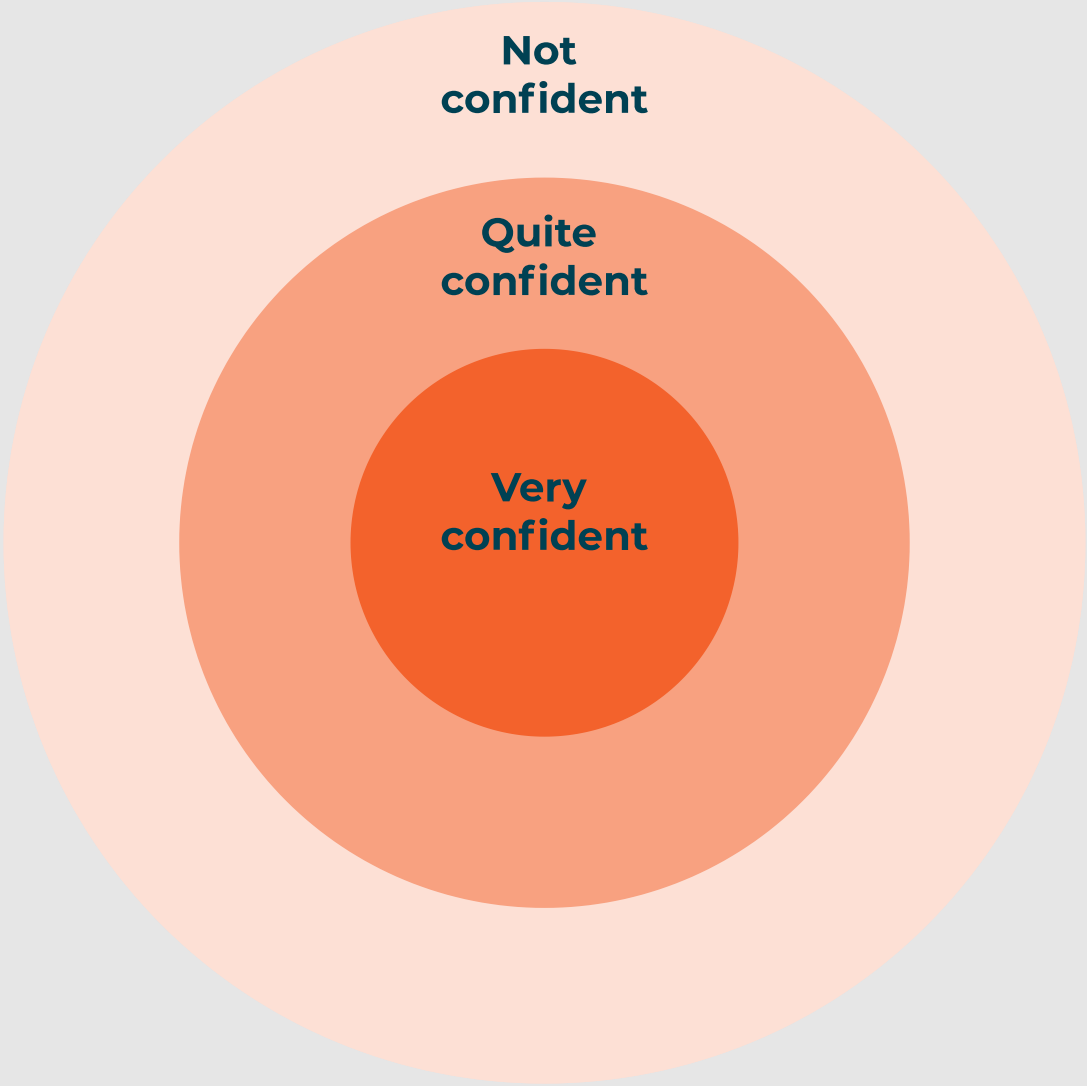
```
result1 = list(frange(0, 3.5, 0.25))
  result2 = list(frange(3.5))
  if result1 == result2:
    print("Default worked!")
  else:
    print(f"Oops! {result1} not equal to {result2}")
```

Advanced Collections – stretch exercises for Kings

3

Did you notice any inaccuracies in the numbers in the previous exercises? This is expected and is the nature of using floating points. They are so serious that this code should not be used in a production environment. A solution is to use the decimal module from the Python Standard library. This has a decimal.Decimal class constructor that converts integers/strings to Decimal objects - do this for all the parameters (start, stop and step) AND then convert the value to be yielded back to a float.

Modify your script 'C:\labs\gen.py' and frange() function to use the decimal module. The test results should be more sensible! Test using same calls in question 4.



Back to the quest

Packages, Modules, Namespaces

Let's take CODE REUSE up a level

- **Modularity**
- **Building a Package**
- **Is is a program or a module?**
- **Namespaces - one honking great idea!**
- **The __main__ namespace trick**
- **Profiling**
- **Doc Testing**
- **Unit Testing**

Modules and Packages

Modular programming is the process of breaking down a large complex problem into separate, smaller and more manageable subtasks or modules. These simpler individual modules can then be completed (possibly in parallel for larger teams) and then glued back together like building blocks.

There are several advantages to using a modularised approach including:
- **Simplicity**: a module typically focuses on one small part of the problem and is easier to design and code.
- **Maintainability**: module are independent and designed, coded, debugged and tested and modified with impacting others.
- **Reusability:** modules can be reused across projects and programs. The 'Holy Grail'!
- **Scoping**: modules can have a separate namespace (honking great idea) avoiding conflicts with identifier names.

Python supports **Packages**, **Modules** and **Functions** that all promote the concept of modularisation.

There are three ways to define a module to be used in Python:
1. It can be written in Python itself with a .py suffix. This is the most common and the one we'll focus on.
2. It can be written in C and loaded dynamically at run-time. The re module uses this!
3. A built-in module is intrinsically contained with the compiler, e.g., built-ins.

Python Package
Technically, a Python Package is just another module with sub-modules, sub-packages and a __path__ attribute. So importing a module or package is really just the same. But if you are looking for a definition - then a regular Package is NAMED directory with a logical group of files, modules, sub-folders AND a file called __init__.py. If the __init__.py is missing then its called a Namespace package.

In general, sub-modules and sub-packages are not imported when a package is imported. The __init__.py can be used to include all or any sub-modules or sub-packages. It can also contain global constants and functions and can also be empty!

Python automatically stores compiled python modules in a **__pycache__** folder which is found in the Package directory or Python install directory. When you are importing a module, including Python Standard Library modules, it is the pre-compiled modules with a suffix .pyc which are loaded, reducing the time to compile. If the .py module file has a newer timestamp then it is re-compiled and copied back in to the __pycache__ folder.

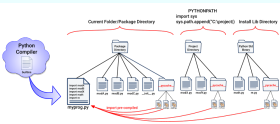
Finding Modules

The Python interpreter finds the modules – as a built-in, in the current or package directory, in a list of directories defined in the PYTHONPATH environment variable, or an installation-dependent list of directories configured during Python install.

On Linux, $ export PYTHONPATH="${PYTHONPATH}:${HOME}/python/lib:/projectX/lib"
On Windows, [Right Click]Computer > Properties >Advanced System Settings > Environment Variables >System Variables >New

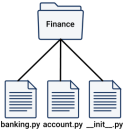
Within a program,
```
>>> import sys
>>> sys.path.append(r"C:\labs\projects\Proj_X")
>>> print(sys.path)
```

Importing a Package, and then a sub-package (from the current package),
```
>>> import my_package
>>> from . import my_sub_package
```



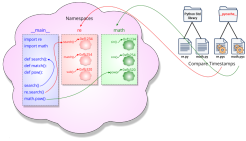
Creating Package

Packages are a logical group of modules that share the same directory and namespace and help avoid naming collisions using the **dot notation** – in a similar way that hierarchical file systems can have files with the same names but with a different pathname. Packages can be created and named in Pycharm, and a new directory will be created with the __init__.py.



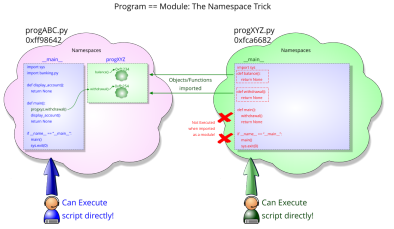

```
#! /bin/python
# Name:         banking.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This module defines several functions for banking app.
"""
    Manages Bank accounts with deposits and withdrawals.
"""
import sys

def deposit(balance):
    """ Deposit monies into bank account """
    amount = float(input("Please enter amount to be added: "))
    balance += amount
    print(f"Deposited {amount}.")
    return balance

def withdraw(balance):
    """ Withdraw monies from bank account """
    amount = float(input("Please enter amount to withdraw: "))
    balance -= amount
    print(f"{amount} withdrawn.")
    return balance

def main():
    """ Withdraw and deposit monies """
    bank_balance = 0
    print(f"Welcome to mBA (mini Bank Account) App")

    while True:
        menu = f"""
                mBA Menu
                --------
                Current Balance {bank_balance}
                1. Deposit monies.
                2. Withdraw monies.
        """
        print(menu)
        option = input("Enter option (1-2, q=quit): ")
        if option == "1":
            bank_balance = deposit(bank_balance)
        elif option == "2":
            bank_balance = withdraw(bank_balance)
        elif option.lower() == "q":
            break
        else:
            print("Invalid option")

    print("Thanks for using the mBA app.")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Packages & Namespaces




```
#! /bin/python
# Name:         account.py
# Author:       OA2.0, Donald Cameron
# Revision:     v1.0
# Description: This is MBA - Minions Banking App used by millions of minions!.
"""
    Withdraw, Deposit and Display banking information.
"""

import sys
import banking
# from bank_account import *    # TRY THIS!

def display_info(bal, scode):
    """ Display bank account info """
    print(f"Current balance = £{bal:.2f}, sort code = {scode}")
    return None

def main():
    """ Withdraw, deposit and display bank account information """
    bank_balance = 34_500.23
    sort_code = "80-45-37"

    print("Welcome to MBA - Minions Banking App")
    name = input("Enter your name: ").title()
    while True:
        menu = f"""
            Menu Options
            ------------
            1. Display Balance
            2. Deposit monies
            3. Withdraw monies
        """
        print(menu)
        option = input("Enter option (1-3, q=quit): ")
        if option == "1":
            display_info(bank_balance, sort_code)
        elif option == "2":
            bank_balance = banking.deposit(bank_balance)
        elif option == "3":
            bank_balance = banking.withdraw(bank_balance)
        elif option.lower() == "q":
            break
        else:
            print("Invalid option")

    print(f"Thank you {name} for using our app!")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Namespace Trick



Have you noticed the **__name__ == "__main__"** test we have been using in conjunction with a main() function? It's called the namespace trick and allows your script to act as a module and your module to act as a script. In other words, it allows the user to execute the module directly as a script, and also import the module into another script without executing the main() function!

It's also part of structuring our python scripts correctly. Continue using it! Oh, and always end your script with an error code, 0 indicates zero errors and any other integer (1-255) indicates an error code! This exit code is passed back to the calling program and indicates success or failure of your script.

Importing - the different ways

`import bank_account` `bank_account.deposit()`	Safer and traceable
`import sys, re, bank_account`	PEP8 disapproves!
`import sys` `import re` `import bank_account`	PEP8 approves!
`import bank_account as b_acc` `b_acc.deposit()`	Module alias
`from bank_account import *`	BANG! Namespace Collision
`from bank_account import deposit`	Safer, be careful!
`from bank_account import (desposit as dp)` `dp()`	Confusing

Profiling

Performance and quality

```
#! /bin/python
# Name:         demo_profiling_before.py
# Author:       GAD_0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to benchmark/profile code
# to improve performance and quality using the cProfile module.
"""
    Search a given file using regular expressions and display
    matched lines. Analyse performance of code change.
"""
import sys
import re

def search_pattern(pattern, file="C:\labs\words"):
    """ Display lines in a given file which match 4_Regular_Expressions """
    fh_in = open(file, mode="rt")

    for line in fh_in:
        m = re.search(pattern, line) # Return None or RE Match object.
        if m:
            print(line, end="")
    fh_in.close()
    return None

def main():
    """ Demonstrate different 4_Regular_Expressions patterns """
    search_pattern(r"^.{19}$") # Match lines with exactly 19 chars.
    return None

if __name__ == "__main__":
    import cProfile
    cProfile.run('main()') # Display stats to console.
    # cProfile.run('main()', 'stats.prof') # Write stats to file.
    sys.exit(0)
```

Recompile pattern for each line in file!

...analyse before

```
#! /bin/python
# Name:         demo_profiling_after.py
# Author:       GAD_0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate how to benchmark/profile code
# to improve performance and quality using the cProfile module.
"""
    Search a given file using regular expressions and display
    matched lines. Analyse performance of code change.
"""
import sys
import re

def search_pattern(pattern, file="C:\labs\words"):
    """ Display lines in a given file which match 4_Regular_Expressions """
    fh_in = open(file, mode="rt")
    robj = re.compile(pattern)

    for line in fh_in:
        m = robj.search(line) # Return None or RE Match object.
        if m:
            print(line, end="")
    fh_in.close()
    return None

def main():
    """ Demonstrate different 4_Regular_Expressions patterns """
    search_pattern(r"^.{19}$") # Match lines with exactly 19 chars.
    return None

if __name__ == "__main__":
    import cProfile
    cProfile.run('main()') # Display stats to console.
    # cProfile.run('main()', 'stats.prof') # Write stats to file.
    sys.exit(0)
```

Recompile pattern ONCE before reading from file.

...analyse after

The Python Standard Library module **'cProfile'** can be used to understand the way your code behaves and is especially useful with large and complex programs. In its simplest form, the output generated will tell you how many calls each of your functions and methods received, and how long was spent executing each function (by line number and file/module name).

The analysis can be displayed to screen or written to a statistics file using a second parameter for the cProfile.run() function.

To analyse the statistics file:

```
C:\labs> python -m pstats stats.prof
% sort cumtime
% stats
```

Doc Testing

Automated doc testing

The **doctest** module searches for text that looks like interactive Python sessions in the Docstrings, and then executes those sessions to verify that they work as shown. This is useful for checking that a module's Docstrings are up to date, for regression testing when code changes and to embed a tutorial within the documentation.

Test in interactive Python session.

```
>>> add(4, 3)
7
```

Copy and paste into function Docstring.

```
#! /bin/python
# Name:        demo_doctesting.py
# Author:      GA2.0, Donald Cameron
# Revision:    v1.0
# Description: This program will demonstrate another example of
# creating functions with DocStrings, parameter passing, return values.
"""
    Calculator program with Add, Divide and Multiply functionality.
"""
import sys

def multiply(x, z):
    """
    Accepts two numeric parameters and return product.
    >>> multiply(4, 3)
    12

    :param x (int|float): Integer or float
    :param z (int|float): Integer or float
    :return (int|float): Numeric product of x and z
    """
    return x * z

def add(x, z):
    """
    Accepts two numeric parameters, adds them and returns sum.
    >>> add(4,3)
    7

    :param x (int|float): Integer or float
    :param z (int|float): Integer or float
    :return (int|float): Numeric of x summed with z
    """
    return x + z

def divide(x, z):
    """
    Accepts two numeric parameters, divides first by the second and returns result.
    >>> divide(4, 3)
    '1.333'
    >>> divide(4, 0)
    Traceback (most recent call last):
    ZeroDivisionError: division by zero

    :param x (int|float): Integer or float
    :param z (int|float): Integer or float
    :return (str): f-string of x divided by z to 3 decimal places.
    """
    return f"{x/z:.3f}"

def main():
    """ Single Line docstrings are ideal if the function is self explanatory """
    print(f"4 * 3 = {multiply(4, 3)}")
    print(f"4 + 3 = {add(4, 3)}")
    print(f"4 / 3 = {divide(4, 3)}")
    # print(f"4 / 3 = {divide(4, 0)}")

    print(multiply.__doc__)
    return None

if __name__ == "__main__":
    import doctest
    doctest.testmod()
    main()
    sys.exit(0)
```

You can have multiple usage examples in your Docstring.

Run the embedded Docstring tests

Unit Testing

Unit testing your code is an essential part of the software development life cycle. As code is written or modified, it must be tested before deployment and existing code must be retested (regression testing) to ensure that the software still behaves as expected (i.e., no unwanted side effects).

Most programmers have done some form of testing, with exploratory testing the most natural. Exploring the code to find bugs, issues, and invalid inputs, but without a plan. Manual testing is more formal, as it incorporates a list of all the features to test, different inputs and expected results. And every time a change is made, we have to go manually go through the list again. And the fun ends!

Automated testing is more thorough and dependable with far better results. In this case, the plan of tests is scripted. Python comes with a set of tools and libraries to help create automated tests. Testing is a huge topic in programming and Python but we will learn to create and execute a basic test using the Python Standard Library **unittest** module.

Unit testing, as the name implies is about a smaller test that checks a single component. If you are testing multiple components then that is **integration testing**. In unit testing, when we are calling a single function with parameters, this is called a test step. And when we are checking the results, this is called a test assertion. Python already has an **assert()** function.

```
>>> assert sum([10, 20, 50]) == 80, "Should be 80"  # Returns an AssertionError and "Should be 80" if fails.
>>> assert sum([1, 1, 1]) == 3, "Should be 3" # Returns an AssertionError and "Should be 80" if fails.
```

This can then be incorporated in function code:

```
from demo_calculator import add, multiply
def test_sum():
    assert add(10, 20) == 30, "Should be 30"

def test_multiply():
    assert multiply(10, 20) == 200, "Should be 200"

if __name__ == "__main__":
    test_sum()
    test_multiply()
```

This is ok for simple checks, but for multiple failures we need a test runner which is a special app designed for running tests and validating output etc. The **unittest** module has both a **test framework** and **runner**. The initial difference is we put our tests into classes as methods (which we introduce in the next session) and use special assertion methods from the module rather than the built-in assert function.

```
#! /bin/python
# Name:         demo_unittest_calc.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This program will demonstrate another example of
# creating functions with DocStrings, parameter passing, return values.
"""
    Calculator program with Add, Divide and Multiply functionality.
"""
import unittest
from demo_calculator import add, multiply, divide

class TestCalculator(unittest.TestCase):
    def test_add(self):
        self.assertEqual(7, add(4, 3))
        return None

    def test_multiply(self):
        self.assertEqual(12, multiply(4, 3))
        return None

    def test_divide(self):
        self.assertEqual('1.333', divide(4, 3))
        return None


if __name__ == "__main__":
    unittest.main()
```

Quiz

Packages, Modules and Namespaces Knowledge Check

5-10 mins

1. Name the command used to load a Python module?

To be revealed

2. Fill in the blank.

Python modules have a To be revealed suffix?

3. How does the Python compiler know when to recompile a Python Standard Library module?

?

a. Module Name

b. Timestamps

c. File size

d. Importing using: from math import sin

4. Given a module called banking.py which has a function called deposit().

Choose from the following, the code which would work:

?

a. from banking.py import deposit
deposit(100)

b. import deposit from banking
banking.deposit(200)

c. import banking
banking.deposit(300)

d. from banking import deposit
banking.deposit(400)

5. Name the file which is created when a new Package is created?

To be revealed

6. True/False? Does this statement satisfy PEP8 requirements?

```
>>> import sys, math, banking
```

To be revealed

Exercises

30 mins

Move your piece down as you complete the exercises

Modules and packages – exercises for 'brave' Knights

1

In this exercise, we are going to pull together some of the scripts and functions that you have already written and load them in as modules to a top-level menu program.

Create a new script **'C:\labs\menu.py'** that imports the appropriate functions from earlier lab exercises.

The script should display a menu like the following diagram, be contained within an infinite loop, and have a mechanism to quit out of the menu and the script. You should only import the functions that are required, which means you may have to modify the modules so that they have appropriate separate functions (and not solely a main() function).

Modules and packages – stretch exercises for Kings

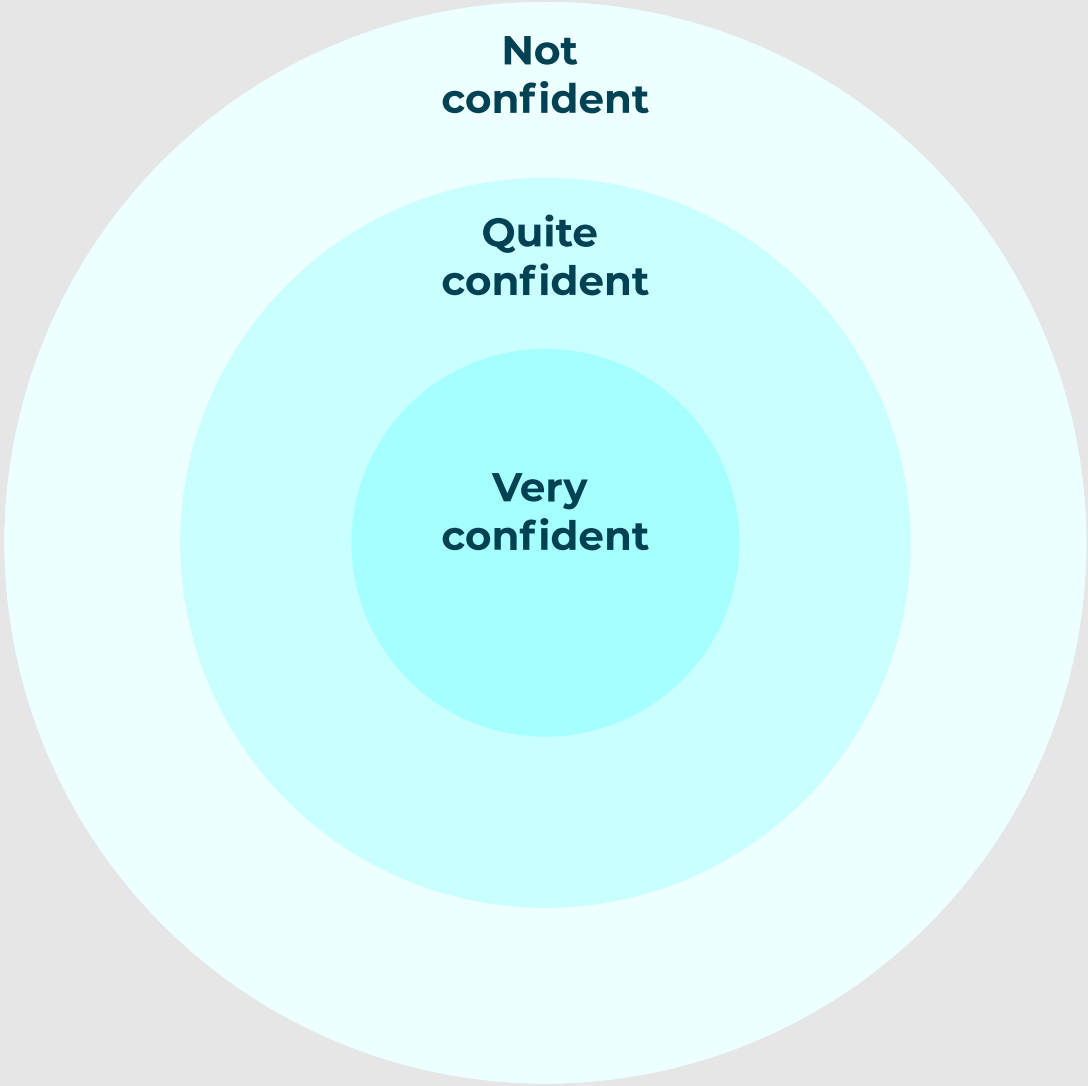
2

Sadly even Kings, when looking for the holy Grail, can be arrested by the police. So we are going to really stretch you in this exercise. Write a new module called **'c:\labs\police_data.py'**. Create two functions, once called **getPoliceForces()** which returns a list of Police Forces in England, and the other **getCrimeByLoc()** which returns a list of Crimes by location (given as latitude/longitude).

This data can be accessed from "https://data.police.uk/api/" by sending **HTTP GET** requests. The Python standard library requests module can be used to send the HTTP GET requests.

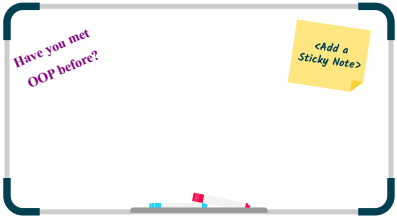
Some sample code for each function is below and https://data.police.uk/docs/ for help. Don't worry if you run out of time, this could be one for the evening! Extra Kudos for exception handling. [Hint: the FUN demos might help here]

```
base_url = r"https://data.police.uk/api/"

full_url = base_url + "forces"
force_data = requests.get(full_url)

full_url = base_url + "crimes-at-location?lat=" + lat + "&lng=" + lon + "&date=" + date
crime_data = requests.get(full_url)
```



Back to the quest

- Benefits of OOP
- Defining a Class
- Getter/Setter methods
- Encapsulation
- Inheritance
- Decorators
- Instantiation
- Duck Typing
- Properties
- Polymorphism



OOP

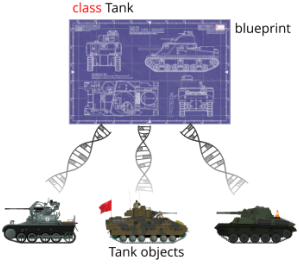
Object-oriented programming (OOP) is a computer programming model of structuring a program by bundling related properties and behaviours into individual objects.

In traditional or procedural programming, if we wanted to write a game with tanks, jeeps, and trucks (as well as buildings, trees, and lakes) we would have to describe everything as a separate variables and each thing would have to have a collection of functions describing what each thing can do. So tank1 would have a variable (data) and lots of functions (tank1_accel, tank1_decel, tank1_rotate_left etc). And we would have to replicate this for all tanks (and all other things). And we end up with a rather large source code file/s and lots of replicated code.

OOP recognises that in all walks of life (and programming) there is a lot of replication, for example, bank accounts, chess boards, online games etc. So in OOP, we create one blueprint called a **class** for each thing, such as a bank account, a pawn or a tank. And the class acts as a template for creating similar **objects**, but these objects are created at runtime in memory (when they are needed) and the source code file remains smaller and simpler.

You have already seen that Python has built-in classes such as str, int, list, dict and set. Each of these has attributes (data) and things it can do to itself (methods). The type() function tells you what type an object is and the dir() function displays a directory of the attributes and method names. And in your program you can create as many instances of str objects as you want based on the str class.

Python provides tools for creating our classes (of anything) with data and methods. And we can instantiate as many objects of this class as we need. In our demonstrations, we are going to bypass bank accounts and chess pieces and create Tank objects!



Defining classes

A few tips when creating a class:
· A class is declared using class, and membership is by indentation.
· "class names use the CapWords convention" – PEP008.
· Methods are declared as functions within that class, the first argument passed is the object.
· The constructor, called when a new object is created is called __init__
· The destructor is called __del_ but it's rarely required and unreliable.
· Classes are usually declared in a file with the same name as the class, with .py appended
· A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition.

```
#! /bin/python
# Name:        tank.py
# Author:      GAS B, Donald Cameron
# Revision:    v1.0
# Description: This module describes a class of Tank for
# an online game.

"""
    Derived class of Tank
"""

class Tank:

    __game_version = "1.0.42" # Global class variable.

    def __init__(self, country, model):
        self.country = country
        self.model = model
        self._speed = 0
        self._direction = 0
        self._location = {'x':0, 'y':0, 'z':0}
        self._shells = 20
        self._health = 100
        # Implicitly called so we do not return.

    def accel(self, increase):
        self._speed += increase
        return None

    def decel(self, decrease):
        self._speed -= decrease
        return None

    def rotate_left(self, degrees):
        self._direction -= degrees % 360
        return None

    def rotate_right(self, degrees):
        self._direction += degrees % 360

    def shoot(self):
        self._shells -= 1
        return None

    def take_damage(self, amount):
        self._health -= amount
        return None

    #### EXTRE CODE for SPECIAL METHODS ####

    # ESample of Operator overloading.
    def __add__(self, other):
        return self._health + other._health

    # Example of Duck Typing, Quack like a str object.
    def __str__(self):
        return f"Health={self._health}, speed={self._speed}, shells={self._shells}"

    # Example of a GETTER method.
    def get_health(self):
        return self._health

    # Example of a SETTER method.
    def set_health(self, newhealth):
        self._health = newhealth
        return None

    #### EXTRA CODE for special PROPERTIES. ####

    # Property function is wrapping the two methods with one name/interface!
    tank_health = property(get_health, set_health)

    #### EXTRA CODE for DECORATORS. ####

    # Wrapping the Identical Named Methods with a DECORATOR to indicate when
    # they should be used. DECORATORS = PYTHONIC!
    @property
    def tank_health(self):
        return self._health

    # Example of a SETTER method.
    @tank_health.setter
    def tank_health(self, newhealth):
        self._health = newhealth
        return None

    @classmethod
    def get_version(cls):
        return f"Game version {cls.__game_version}"
```

Extra code to be added during DEMO by trainer

Instantiation

Once the class has been created (usually in its own module file). We then need to import the class and create **instances** of the class - in this case, we need to 'spawn' new tank **objects** in memory.

Once an instance of an object has been created, that object **inherits** the attributes (data) and behaviour (methods) of the class its based on. If any of the attributes or methods have an underscore prefix in their name, it identifies them as private and local to the class/object and should not be accessed outside the object. This is called **encapsulation** (hidden in a capsule). Using a double underscore prefix mangles the identifier by prefixing the name with _modulename.

Attributes and methods are accessed in the same way we access methods for built-in Python objects, using the dot notation, for example str.upper().

Another important features of OOP is **polymorphism** (many forms) that allows us to perform a single action in different ways. For example, we could have Tank, Jeep and Helicopter objects which all have a shoot() method - but each shoots in a different way!

We can use the built-in functions type() and dir() to get information on a Tank object and isinstance() can check whether an object belongs to a class. For example, isinstance(robin_tank, Tank).

```
#! /bin/python
# Name:         instances.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This is an ultra realistic computer game with Tanks!
"""
    Game of Tanks!
"""

import sys
import tank

def main():
    """ Main game """
    # Instantiate 3 new Tank objects.
    lancelot_tank = tank.Tank("German", "Tiger")
    arthur_tank = tank.Tank("British", "Churchill")
    robin_tank = tank.Tank("American", "Sherman")

    # .. and the game begins.
    lancelot_tank.accel(41)
    arthur_tank.accel(33)

    robin_tank.rotate_left(385)
    robin_tank.accel(15)
    robin_tank.shoot()

    # ..and success!
    lancelot_tank.take_damage(40)
    arthur_tank.take_damage(62)

    # And now for some game visuals! Well at least a print statement!
    print(f"Health of Lancelot's Tank = {lancelot_tank._health}") # Why is this
    # POOR Code?

    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Special methods

In Python you will find special methods which start and end with double underscores. They are called **Magic methods** or **dunders**. They are not meant to be called directly are invoked internally from the class when a certain action occurs, For example, when you add two objects of the same class using the '+' operator, then internally the __add__() method will be called.

When we create a new object based on a class, the __new__() method is called immediately followed by the __init__() to initialise it. We rarely have to override __new__() in our class definitions, but we usually always override __init__() to initialise new objects.

Some more magic methods that can be overridden:

__len__(self)	Implement the len() function
__str__(self)	Returns a string object when the object is converted into a string.
__add__	+ Object can be used with 'add' operator.
__sub__	- Object can be used with 'minus' operator.
__eq__	== Object can be compared using the equality operator.
__ge__	>= Object can be compared using the 'greater or equal' operator.
__lt__	< Object can be compare using the 'less then' operator.

Append this special method to tank.py

```
    # Example of Duck Typing, Quack like a str object.
    def __str__(self):
        return f"Health={self._health}, speed={self._speed}, shells={self._shells}"


#! /bin/python
# Name:         demo_special_methods.py
# Author:       GA2.0, Donald Cameron
# Revision:     v1.0
# Description: This is an ultra realistic computer game with Tanks!
"""
    Game of Tanks!
"""

import sys
import tank

def main():
    """ Main game """
    # Instantiate 2 new Tank objects.
    lancelot_tank = tank.Tank("German", "Tiger")
    arthur_tank = tank.Tank("British", "Churchill")
    robin_tank = tank.Tank("American", "Sherman")

    # .. and the game begins.
    lancelot_tank.accel(41)
    arthur_tank.accel(33)

    robin_tank.rotate_left(385)
    robin_tank.accel(15)
    robin_tank.shoot()

    # ..and success!
    lancelot_tank.take_damage(40)
    arthur_tank.take_damage(62)

    # And now for some game visuals! Well at least a print statement!
    # Why is this POOR Code?
    print(f"Health of Lancelot's Tank = {lancelot_tank._health}") # Poor Code

    # Example of Operator Overloading.
    print(f"Status of Lancelot's and Arthur's Tanks = {lancelot_tank + arthur_tank}")

    # Example of Duck Typing, Tank can now Quack like a str!
    print(f"Status of Lancelot's Tank: {lancelot_tank}")

    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Add this code!

Getter & Setter Methods

In Object Oriented Programming we often want to keep the data belonging to an object private and manage what can access or modify our data. For example in a game you would not want other Tank objects knowing of your location and dire health as they would likely sneak up on you. We can limit what data is accessed using the dunder prefix but this is only a convention amongst fellow programmers and not enforced by the language.

The answer is special methods for **accessing** and **mutating** the data, more commonly called **Getter** and **Setter** methods. These methods control access and changes to the objects data, perhaps checking if you're a friendly tank or updating your health if you have received a health boost?

Append these two special methods to tank.py

```
        # Example of a GETTER method.
        def get_health(self):
            return self._health

        # Example of a SETTER method.
        def set_health(self, newhealth):
            self._health = newhealth
            return None
```

```
#! /bin/python
# Name:         demo_getter_setter_methods.py
# Author:       OAJ.D, Donald Cameron
# Revision:     v1.0
# Description: This is an ultra realistic computer game with Tanks!
"""
    Game of Tanks!
"""

import sys
import tank

def main():
    """ Main game """
    # Instantiate 3 new Tank objects.
    lancelot_tank = tank.Tank('German', 'Tiger')
    arthur_tank = tank.Tank('British', 'Churchill')
    robin_tank = tank.Tank('American', 'Sherman')

    # ... and the game begins.
    lancelot_tank.accel(45)
    arthur_tank.accel(33)

    robin_tank.rotate_left(305)
    robin_tank.accel(15)
    robin_tank.shoot()

    # ...and success!
    lancelot_tank.take_damage(40)
    arthur_tank.take_damage(62)

    # And now for some game visuals! Well at least a print statement!
    print(f'Health of Lancelot's Tank = {lancelot_tank._health}') # Why is this POOR Code?

    # Example of Operator Overloading.
    print(f'Status of Lancelot's and Arthur's Tanks = {lancelot_tank + arthur_tank}')

    # Example of Duck Typing. Tank can now Quack like a str!
    print(f'Status of Lancelot's Tank: {lancelot_tank}')

    # Lancelot has received a health boost!
    # Update and get health directly from private objects! Ugly and Poor!
    lancelot_tank._health = 98
    print(f'Lancelot's new health is {lancelot_tank._health}')

    # Examples of special GETTER and SETTER methods.
    # Internal methods accessing private data = better!
    lancelot_tank.set_health(100)
    print(f'Lancelot's new health is {lancelot_tank.get_health()}')

    return None

if __name__ == '__main__':
    main()
    sys.exit(0)
```

Add this code to test the `__str__` method (Duck Typing)

Property function

Getter and Setter methods are good for managing access to private data within an object, but you likely have to use multiple different names for them and parameters are passed within parentheses.

As an alternative, Python has a built-in **property()** function that when applied to a method, it makes it behave like a variable interface with access, assignment and deletion properties (and an optional documenting property). Or in OOP speak, with getter, setter, and deleter actions.

For example, if we want to access the health of a tank, assign or delete a value to the health of a tank using just one interface/variable name:

tank_health = property(get_health, set_health, del_health, doc_health)

Append this line to tank.py

```
        # Property function is wrapping the two methods with one name/in
        tank_health = property(get_health, set_health)
```

```
#! /bin/python
# Name:         demo_properties.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This is an ultra realistic computer game with Tanks!
"""
    Game of Tanks!
"""

import sys
import tank

def main():
    """ Main game """
    # Instantiate 3 new Tank objects.
    lancelot_tank = tank.Tank("Garman", "Tiger")
    arthur_tank = tank.Tank("British", "Church11")
    robin_tank = tank.Tank("American", "Sherman")

    # .. and the game begins.
    lancelot_tank.accel(41)
    arthur_tank.accel(33)

    robin_tank.rotate_left(385)
    robin_tank.accel(15)
    robin_tank.shoot()

    # ..and success!
    lancelot_tank.take_damage(40)
    arthur_tank.take_damage(62)

    # And now for some game visuals! Well at least a print statement!
    print(f"Health of Lancelot's Tank = {lancelot_tank._health}") # Why is this POOR Code?

    # Example of Operator Overloading.
    print(f"Status of Lancelot's and Arthur's Tanks = {lancelot_tank + arthur_tank}")

    # Example of Duck Typing, Tank can now Quack like a str!
    print(f"Status of Lancelot's Tank: {lancelot_tank}")

    # Lancelot has received a health boost!
    # Update and get health directly from private objects! Ugly and Poor!
    lancelot_tank._health = 98
    print(f"Lancelot's new health is {lancelot_tank._health}")

    # Examples of special GETTER and SETTER methods.
    # Internal methods accessing private data = better!
    lancelot_tank.set_health(100)
    print(f"Lancelot's new health is {lancelot_tank.get_health()}")

    # Using a property interface with one name.
    lancelot_tank.tank_health = 101
    print(f"Lancelot's new health is {lancelot_tank.tank_health}")

    # And testing our class method decorator.
    print(tank.Tank.get_version())
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Add this code to test the property function

Decorators

A Python **decorator** is a **function** that takes another function and extends the behaviour of the latter function without explicitly modifying it. For example, the **@property** decorator is a built-in decorator for the property() function and can use the following three decorators - @property, @<function-name>.setter, @<function-name>.deleter.

All the following three methods have the same name (overloaded) but we define when we use each one by decorating it a property function, one as a getter, one as a setter and the other as a deleter.

```
@property
def tank_health(self):
    return self._health

@tank_health.setter
def tank_health(self, newhealth):
    self._health = newhealth
    return None

@tank_health.deleter
def tank_health(self):
    del self._health
    return None
```

Comment this line

```
    # tank_health = property(get_health, set_health)

    #### EXTRA CODE for DECORATORS. ####

        # Wrapping the Identical Named Methods with a DECORATOR to indicate when
        # they should be used. DECORATORS = PYTHONIC!
        @property
        def tank_health(self):
            return self._health

        # Example of a SETTER method.
        @tank_health.setter
        def tank_health(self, newhealth):
            self._health = newhealth
            return None
```

Append this code to tank.py

```
#! /bin/python
# Name:         game_properties.py
# Author:       GKS A. Donald Cameron
# Revision:     v1.0
# Description: This is an ultra realistic computer game with Tanks!
"""
    Game of Tanks!
"""

import sys
import tank

def main():
    """ Main game """
    # Instantiate 3 new Tank objects.
    lancelot_tank = tank.Tank('German', 'Tiger')
    arthur_tank = tank.Tank('British', 'Churchill')
    robin_tank = tank.Tank('American', 'Sherman')

    # .. and the game begins.
    lancelot_tank.accel(45)
    arthur_tank.accel(22)

    robin_tank.rotate_left(180)
    robin_tank.accel(15)
    robin_tank.shoot()

    # ...and success!
    lancelot_tank.take_damage(66)
    arthur_tank.take_damage(23)

    # And now for some game visuals! Well at least a print statement!
    print(f'Health of Lancelot's Tank = {lancelot_tank._health}') # Why is this POOR Code?

    # Example of Operator Overloading.
    print(f'Status of Lancelot's and Arthur's Tanks = {lancelot_tank + arthur_tank}')

    # Example of Duck Typing. Tank can now Quack like a str!
    print(f'Status of Lancelot's Tank: {lancelot_tank}')

    # Lancelot has received a health boost!!
    # Update and get health directly from private object! Ugly and Poor!
    lancelot_tank._health = 90
    print(f'Lancelot's new health is {lancelot_tank._health}')

    # Examples of special GETTER and SETTER methods.
    # Internal methods accessing private data = better!
    lancelot_tank.set_health(76)
    print(f'Lancelot's new health is {lancelot_tank.get_health()}')

    # Using a property interface with one name.
    lancelot_tank.tank_health = 50
    print(f'Lancelot's new health is {lancelot_tank.tank_health}')

    # And testing our class method decorator.
    print(tank.Tank.get_version())
    return None

if __name__ == '__main__':
    main()
    sys.exit(0)
```

Re-run game.py with no changes.

Inheritance

Have you noticed that some objects in the world have got similar attributes and behaviours? For example, jeeps and trucks do similar things as tanks such as accelerate, decelerate, and turn left and right. Current accounts have got similarities to deposit and savings accounts. In chess pawns, rooks, knights have all got similar properties like colour, position on board, and have similar behaviours such as move forward or move back.

This could lead to replication of data and methods and you would be 'a very naughty boy' in the OO world if you did this. So OOP supports the concept of Inheritance, and indeed multi-inheritance, where an object can be **Derived** from one or more **Base** or **Parent** classes, and will inherit the data and methods from the Base class.

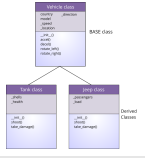
The derived or child class has all the methods and properties of the parent class, but you can add or change them.

For example, we could have a Vehicle base class, and Tanks, Jeeps and Trucks could all inherit from this one parent - thus reducing the replication of code and effort.

You might also find a class that is almost what you need, but you can't change it. If you want a string class that's slightly different from Python's str class, you can't actually change Python yourself, but could inherit from str and add new methods.

In Python, it is very common to derive our own classes from Python classes.

A useful method for classes is **issubclass()** which checks whether a class is a subclass of another class.




```
#! /bin/python
# Name:         vehicle.py
# Author:       QA2.0, Donald Cameron
# Revision:     v1.0
# Description: This module describes a
# generic Vehicle class.
"""
    BASE class of Vehicle
"""

class Vehicle:
    def __init__(self, country, model
        ):
        self.country = country
        self.model = model
        self._speed = 0

    def accel(self, increase):
        self._speed += increase
        return None

    def decel(self, decrease):
        self._speed -= decrease
        return None
```

Change code in tank.py for single inheritance.

Quiz

OOP, Classes and Objects Knowledge Check

5-10 mins

1. Fill in the blank.

In OOP, a class is a **To be revealed** for an object.

In a Python class definition, functions are called?

?

a. functions

b. methods

c. accessors

d. attributes

3. Which is the correct way to instantiate a new knight based on this class?

```
class Knight:
def __init__(self, colour, bravery):
        self.name = f"{colour} knight"
        self.bravery = bravery
```

?

a. Knight()

b. Knight("Green", 4, "pepperami")

c. Knight("Black", 10)

d. Knight.__init__("Green", 4)

4. True or False - a class can inherit from multiple base classes?

To be revealed

Exercises

35 mins

Move your piece down as you complete the exercises

OOP – exercises for stupidly brave Black Knights

1

In this exercise you are going to open an existing module called 'c:\labs\movieDB.py' which will contain one simple class called MovieDB. It will have the following attributes and methods:

We have provided some of the code already for creating the class, opening and closing the DB connection and created a new table if required. The DB is a SQLite3 database called "C:\labs\movie_db_sqlite"

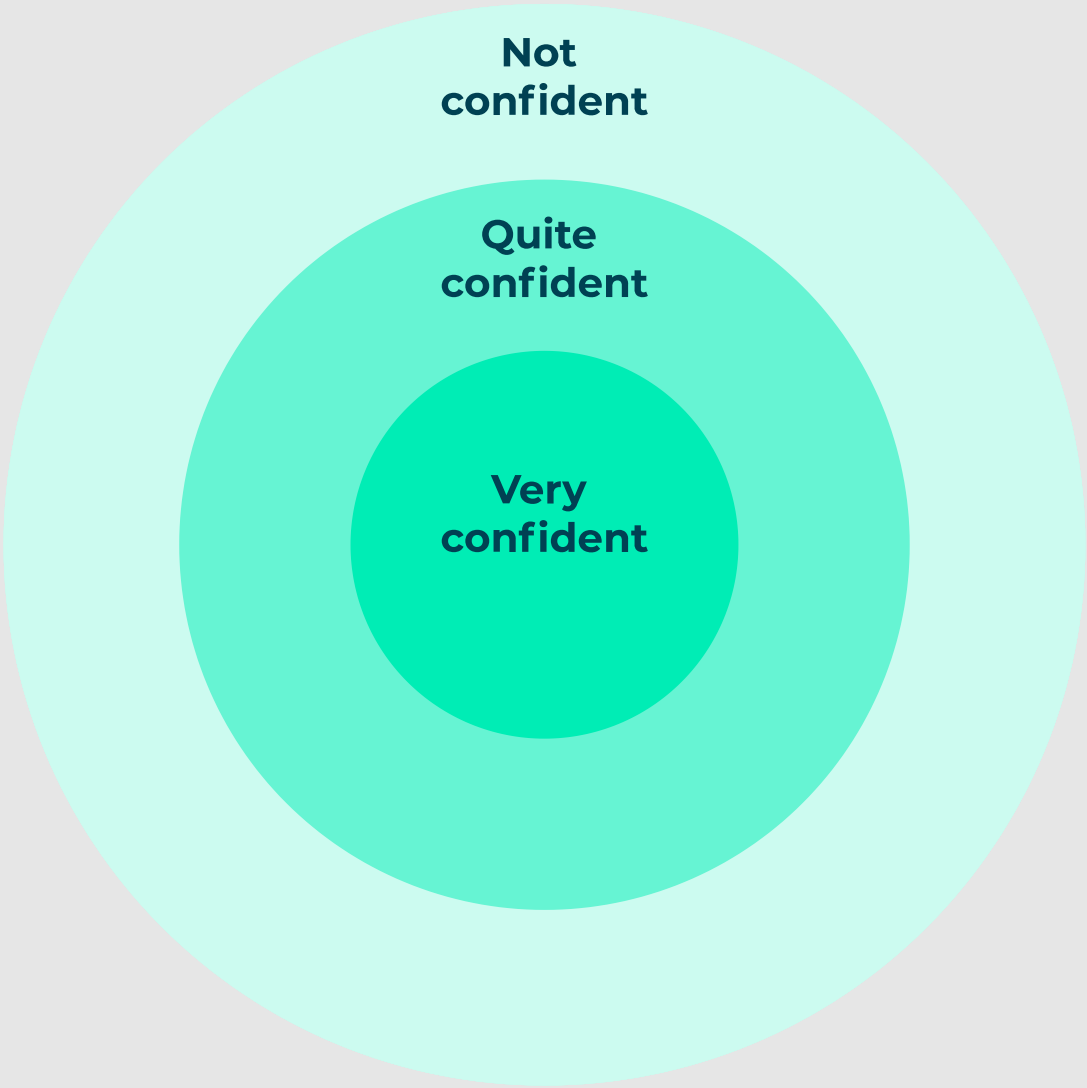
You have to create new methods for inserting new rows, delete existing rows, query all rows and commit all changes.

[Hint: you will find the table definition in an existing method]

```
#! /usr/bin/python
# Author: QALO LIVE_V1.0
# Description: This module defines a class called MovieDB for creating objects
# that can interact with a SQLite3 database file.
"""
 MovieDB class.
"""
import sqlite3

class MovieDB:
 # Class variable storing location of DB
   DB_LOC = "C:\labs\wmovie_db.sqlite"

   def __init__(self):
      try:
        self.__db_conn = sqlite3.connect(MovieDB.DB_LOC)
        self.cur = self.__db_conn.cursor()
      except Exception as err:
        raise ValueError
      # No return as not explicitly called.


   def create_table(self, table_name="movies"):
     self.cur.execute("""CREATE TABLE IF NOT EXISTS movies
                  ( id       INTEGER PRIMARY KEY
                   ,title    VARCHAR(30)
                   ,year     VARCHAR(30)
   ,rating INTEGER
                  )""")
     return None

   def commit(self):
     """ Commit changes to database """
     self.__db_conn.commit()
     return None

   def __del__(self):
     """ Close connection automatically when object is deleted """
     self.__db_conn.close()
     return None
```

2

Write a simple program to import and test the MovieDB class.



Back to the quest

Solutions

5 mins

Quiz

1. Name three ordered and two unordered collections?

Ordered?

str, tuple, list

Unordered

dict, set

2. Is a tuple mutable?

True ✗ False

3. The joy of sets! What would be printed from the following code?

```
>>> scottish_movies = {'Gregory's Girl', 'Local Hero', 'Comfort and Joy', 'That Sinking Feeling'}
>>> fav_movies = {'The Hobbit', 'Local Hero', 'The Lord of the Rings'}
>>> print(scottish_movies & fav_movies)
```

{'Local Hero'}

4. Fill in the blanks:

Dictionaries have unique ____ Keys

Sets have unique ____ Objects

5. What would be printed from the following code?

```
>>> movies = {'Gregory's Girl', 'Local Hero', 'Comfort and Joy', 'That Sinking Feeling'}
>>> fv_fv = movies
>>> print(fv)
```

{'Local Hero', 'Comfort and Joy'}

6. Write the print statement from question 5 using set methods?

print(scottish_movies.intersection(fav_movies)

1

```
#! /bin/python
# Name:        Lotto.py
# Author:      GA2.0, Donald Cameron
# Revision:    v1.0
# Description: Generates 6 Random Unique Lottery numbers.
"""
    Lottery number generator. Good luck and remember your
                    training!
"""

import sys
import random

def main():
    """ The Main Program """
    lotto = set() # Did we mention we want Unique numbers!
    while len(lotto) < 6:
        num = random.randint(1, 56)
        lotto.add(num)

    print(f"Lottery numbers = {lotto}")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

2

```
#! /bin/python
# Name:        topTen.py
# Author:      GA2.0, Donald Cameron
# Revision:    v1.0
# Description: Display top movies.
"""
    Top movies and Rankings.
"""
import sys

def main():
    movies = []

    fh_in = open(r"C:\labs\top250_movies.txt", mode
                 ="rt")
    movies = fh_in.readlines()

    fh_in.close()

    for rank, movie in enumerate(movies, start=1):
        print(f"{rank:>6d}: {movie}", end="")
        if rank == 10: break

    print(f"First movie = {movies[0]}", end="")
    print(f"Last movie = {movies[-1]}", end="\n")

    topN = int(input("Choose Top-N movies: "))
    for rank, movie in enumerate(movies, start=1):
        print(f"{rank:>6d}: {movie}", end="")
        if rank == topN: break
    return None


if __name__ == "__main__":
    main()
    sys.exit(0)
```

3-7

```
#! /bin/python
# Name:        topTen_stretch.py
# Author:      GA2.0, Donald Cameron
# Revision:    v1.0
# Description: Display top movies.
"""
    Top movies and Rankings.
"""
import sys

def get_movies():
    fh_in = open(r"C:\labs\top250_movies.txt", mode
                 ="rt")
    movies = fh_in.readlines()
    fh_in.close()
    return movies

def display_movies(movies, topN):
    for rank, movie in enumerate(movies, start=1):
        print(f"{rank:>6d}: {movie}", end="")
        if rank == topN: break
    return None

def search_movies(text, movies):
    for rank, movie in enumerate(movies, start=1):
        if text in movie:
            print(f"{rank:>6d}: {movie}", end="")
    return None

def main():
    movies = []
    while True:
        menu = """
               Menu Options
               ------------
               1. Get online movie ranking from IMDB
               2. Display top ranking movies
               3. Search for movie
               """
        print(menu)
        option = input("Enter option: ")
        if option == "1":
            movies = get_movies()
        elif option == "2":
            topN = int(input("Choose topN movies:"))
            display_movies(movies, topN)
        elif option == "3":
            text = input("Enter movie to search:")
            search_movies(text, movies)
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Solutions

5mins

Quiz

1. What is the default mode for the built-in open() function?

"rt" [Read and Text mode]

2. What is the default buffering mode for text files?

Line buffering

3. Which methods are used for random access?

.seek() and .tell()

4. Which Python standard library module preserves ONE Python object to a file?

pickle.py

5. Which Python standard library module preserves MULTIPLE Python objects to a file?

shelve.py

1-4

```python
#! /bin/python
# Name:        display_movies.py
# Author:      QA2.0, Donald Cameron
# Revision:    v1.0
# Description: Display movie information
"""
    IMDB movie information
"""
import sys

def main():
    # Solution for Ques 2 & 3.
    fh_in = open(r"C:\labs\top250_movies.txt", mode="rt")
    for movie in fh_in:
        print(f"{movie.title()}", end="")
    fh_in.close()

    # Solution for Ques 4.
    fh_in = open(r"C:\labs\top250_movies.txt", mode="rt")
    for rank, movie in enumerate(fh_in, start=1):
        print(f"{rank}. {movie.title()}", end="")
    fh_in.close()

    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

5

```python
#! /bin/python
# Name:        movie_data.py
# Author:      QA2.0, Donald Cameron
# Revision:    v1.0
# Description: Write detailed movie information to file.
"""
    Persistent Movie Information.
"""

import sys

def main():
    movies = [{"title": "The Shawshank Redemption", "director": "Frank Darabont",
        "year": 1994},
              {"title": "The Godfather", "director": "Francis Ford Coppola", "year":
                1972},
              {"title": "The Godfather; Part II", "director": "Francis Ford Coppola",
                "year": 1974},
              {"title": "The Dark Knight", "director": "Christopher Nolan", "year":
                2008},
              {"title": "12 Angry Men", "director": "Sidney Lumet", "year": 1957}
              ]

    fh_out = open(r"C:\labs\top5_movie_info.txt", mode="wt")

    for movie in movies:
        print(f"{movie['year']:>10} - "
              f"Title:{movie['title'].title():<30s}"
              f"Director:{movie['director']:>25s}", file=fh_out)

    fh_out.close()
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

6

```python
#! /bin/python
# Name:        movie_dict.py
# Author:      QA2.0, Donald Cameron
# Revision:    v1.0
# Description: Preserve Movie dictionary to file.
"""
    Preserving Movie information.
"""

import sys
import pickle
import gzip
import pprint

def main():
    movies = [{"title": "The Shawshank Redemption", "director": "Frank Darabont", "year": 1994},
              {"title": "The Godfather", "director": "Francis Ford Coppola", "year": 1972},
              {"title": "The Godfather; Part II", "director": "Francis Ford Coppola", "year": 1974},
              {"title": "The Dark Knight", "director": "Christopher Nolan", "year": 2008},
              {"title": "12 Angry Men", "director": "Sidney Lumet", "year": 1957}
              ]

    # Open compressed file handle for writing in Binary mode.
    fh_out = gzip.open(r"C:\labs\top5movies.pgz", mode="wb")
    pickle.dump(movies, fh_out)
    fh_out.close()

    # Open compressed file handle for writing in Binary mode.
    fh_in = gzip.open(r"C:\labs\top5movies.pgz", mode="rb")
    films = pickle.load(fh_in)
    fh_in.close()

    pprint.pprint(movies)
    print("-" * 60)
    pprint.pprint(films)
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

7

```python
#! /bin/python
# Name:        display_unused_ports.py
# Author:      QA2.0, Donald Cameron
# Revision:    v1.0
# Description: Display all the unused ports in the SERVICES file.
"""
    Display unused next ports from the SERVICES file.
"""


import sys

def main():
    if sys.platform == "win32":
        filename = r"C:\WINDOWS\System32\drivers\etc\services"
    else:
        filename = r"/etc/services"

    # Open file handle for reading in text mode.
    fh_in = open(filename, mode="rt")
    all_ports = set(range(1, 201))
    used_ports = set()

    # Iterate through file handle and read the services.
    for line in fh_in:
        if line.isspace(): continue
        if line.startswith("#"): continue

        fields = line.split("/") # Split line into list using '/'
        fields = fields[0].split() # Split 1st field using whitespace.
        port = int(fields[1]) # Index 1 is the port number.
        if port <= 200:
            used_ports.add(port)

    print(f"All ports = {all_ports}")
    print(f"Used ports = {used_ports}")
    print(f"Unused ports = {all_ports - used_ports}")

    fh_in.close()

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Solutions

5 mins

Quiz

1. What does B.R.E and E.R.E stand for?

Basic and Extended Regular Expressions

2. What prefix should you use with your Regex patterns?

r"pattern" for Raw pattern strings,

3. Can you think of a better way to write the following pattern, which is 15 chars long?

r"^...............$"

r"^.{15}$"

4. Given this code, what will the pattern match?

```
>>> text = "Brave Sir Robin ran away. Bravely ran away away. When danger
reared it's ugly head, he bravely turned his tail and fled."
>>> m = re.search(r"(brave).*\1", text, flags=re.IGNORECASE)
>>> print(m.group())
```

Brave Sir Robin ran away. Bravely ran away away. When danger reared it's ugly head, he brave

5. Given this code, what would be printed?

```
>>> text = "racecar"
>>> m = re.search(r"^(.)(.)(.).\3\2\1$", text)
>>> print(m.groups(), m.groups()[0], m.group(1))
```

('r', 'a', 'c') r r

1-4

```
#! /bin/python
# Name:        top250.py
# Author:      QA2.0, Donald Cameron
# Revision:    v2.5
# Description: Download the top 250 movies chosen by Letterboxd users.
"""
    Download and display online movie information.
"""
import sys
import requests
from bs4 import BeautifulSoup
import re

def main():
    base_url = "https://letterboxd.com/jack/list/official-top-250-films-with-the-most-fans/page/{}/"
    top_movies = []

    # Scrape the first 4 pages (adjust if necessary)
    for page_num in range(1, 4):  # Adjust range according to the number of pages
        url = base_url.format(page_num)
        response = requests.get(url) # Send a GET request to fetch the page content
        soup = BeautifulSoup(response.text, "html.parser")
        movie_tags = soup.find_all("li", class_="poster-container") # Find all movie containers.

        # Extract movie details
        for movie in movie_tags:
            title = movie.find('img', class_='image').get('alt')
            top_movies.append(title)

    # Display movies  - CODE TO ADDED BELOW.
    pattern = input("Enter movie search string: ")

    for rank, movie in enumerate(top_movies, start=1):
        m = re.search(pattern, str(movie), re.IGNORECASE)
        if m:
            print(f"{rank:>4}: {movie}")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

7

```
#! /bin/python
# Name:        display_unused_ports.py
# Author:      QA2.0, Donald Cameron
# Revision:    v1.0
# Description: Display all the unused ports in the SERVICES
    file.
"""
    Display unused next ports from the SERVICES file.
"""


import sys
import re

def main():
    if sys.platform == "win32":
        filename =
r"C:\WINDOWS\System32\drivers\etc\services"
    else:
        filename = r"/etc/services"

    # Open file handle for reading in text mode.
    fh_in = open(filename, mode="rt")
    all_ports = set(range(1, 201))
    used_ports = set()

    # Iterate through file handle and read the services.
    for line in fh_in:
        # Search for lines only containing digits followed
        # by / and (tcp or udp). Use groupings to extract.
        m = re.search(r"(\d+)/(tcp|udp)", line)
        if m:
            port = int(m.group(1))
            if port <= 200:
                used_ports.add(port)

    print(f"All ports = {all_ports}")
    print(f"Used ports = {used_ports}")
    print(f"Unused ports = {all_ports - used_ports}")

    fh_in.close()
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

5-6

```
#! /bin/python
# Name:        search250.py
# Author:      QA2.0, Donald Cameron
# Revision:    v1.0
# Description: Download the top 250 movies chosen by Letterboxd users.
"""
    Download and display online movie information.
"""
import sys
import requests
from bs4 import BeautifulSoup
import re

def main():
    base_url = "https://letterboxd.com/jack/list/official-top-250-films-with
        -the-most-fans/page/{}/"
    top_movies = []

    # Scrape the first 4 pages (adjust if necessary)
    for page_num in range(1, 4):  # Adjust range according to the number of
        pages
        url = base_url.format(page_num)
        response = requests.get(url) # Send a GET request to fetch the page
            content
        soup = BeautifulSoup(response.text, "html.parser")
        movie_tags = soup.find_all("li", class_="poster-container") # Find all
            movie containers.

        # Extract movie details
        for movie in movie_tags:
            title = movie.find('img', class_='image').get('alt')
            top_movies.append(title)

    # CODE FOR PART 5-6 BELOW
    count = 0
    for rank, movie in enumerate(top_movies, start=1):
        pattern = r"^the" # Movies starting with 'the' = 46 lines.
        # pattern = r"[0-9]"  # Movies with digit in name = 14 lines.
        # pattern = r"[aeiou]{3}"  # Movies with 3 consecutive vowels = 8 lines
            .
        # pattern = r"^(.).*\1$" # Movies starting/ending with same char = 17
            lines.
        # pattern = r"^\d+$"  # 1 line.
        # pattern = r"\s+(\w+)\s+.*\1\s+"  # 4 lines.

        m = re.search(pattern, str(movie), re.IGNORECASE)
        if m:
            print(f"{rank:>4}: {movie}")
            count += 1
    print(f"Matched {count} lines")

    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Solutions

5 mins

Quiz

1. What does (*,) at the beginning of the parameter list do?

Enforce named parameter passing.

2. True of False. Do Python functions always return a value?

True
(Python functions always return None unless explicit value)

2. What is the output of the following function call?

```
>>> def my_func(num):
>>> return num * 10
>>>
>>> my_func(10)
>>> print(num)
```

NameError: name 'num' is not defined

4. What is the output of the following function call?

```
>>> def display_info(*showings):
>>> for x in showings:
>>> print(x)
>>>
>>> display_info('movie','broadcast', 'year='1995')
```

movie
year

1-3

```
#! /bin/python
# Name:       calc.py
# Author:     GAO.B, Donald Cameron
# Revision:   v1.0
# Description: A Calculator program
"""
    Calculator program with add, divide, multiply,
    subtract and modulus functionality.
"""
import sys

def add(x, z):
    """ Return sum of x and z """
    return x + z

def divide(x, z):
    """ Return x divided by z to 3 decimal places
    """
    return x/z

def multiply(x, z):
    """ Return product of x and z """
    return x * z

def subtract(x, z):
    """ Return difference of x and z  """
    return x - z

def modulus(x, z):
    """ Return remainder of x divided by z """
    return x % z

# Solution for Question 7
# def add(*numbers):
#     """ Return sum of all parameters """
#     return sum(numbers)

# def multiply(*numbers):
#     """ Return the product of all parameters """
#     product = 1
#     for num in numbers:
#         product *= num
#     return product

def main():
    """ This is the main program function """
    print(f"11 + 5 = {add(11, 5)}")
    print(f"11 / 5 = {divide(11, 5)}")
    print(f"11 * 5 = {multiply(11, 5)}")
    print(f"11 - 5 = {subtract(11, 5)}")
    print(f"11 % 5 = {modulus(11, 5)}")

    # print(f"11 + 5 = {add(11, 5, 33, 15, 6.5)}")
    # print(f"11 + 5 = {multiply(11, 5, 33, 15, 6.5
        )}")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

4

```
#! /bin/python
# Name:       search250.py
# Author:     GAO.B, Donald Cameron
# Revision:   v1.0
# Description: Download the top 250 movies chosen by IMDb
#       users.
"""
    Download and display online movie information.
"""
import sys
import imdb
import re

def search_movie(pattern):
    """ Search for a Regex pattern in the top 250 movies
        """
    ia = imdb.IMDb()
    count = 0
    for rank, movie in enumerate(ia.get_top250_movies(),
                                 start=1):
        m = re.search(pattern, str(movie), re.IGNORECASE)
        if m:
            print(f"{rank:>4}: {movie}")
            count += 1
    print(f"Matched {count} lines")
    return None

def main():
    """ The Main Program """
    search_movie(r"^The") # # 50 lines.
    search_movie(r"[0-9]") # 11 lines.
    search_movie(r"(mies[)]$") # 5 lines.
    search_movie(r"^(.).+\1$") # 15 lines.
    search_movie(r"^[a-zA-Z]{4}") # 1 line.
    search_movie(r"monty") # 1 lines.
    search_movie(r"(ie|ei)ve.+\1e") # 8 lines.
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

5

```
#! /bin/python
# Name:       searchWords.py
# Author:     GAO.B, Donald Cameron
# Revision:   v1.0
# Description: Search and display lines in multiple files using
# Regular expressions

"""
    Pattern searching Tool
"""
import sys
import re

def search_pattern(pattern, *files):
    """ Search for Regex patterns in multiple files """
    if not files: files = ('C:\\labs\\words',)
    for file in files:
        fh = open(file, mode='rt')
        for line in fh.in_
            m = re.search(pattern, line)
            if m:
                print(line, end='')

        fh.in.close()

    return None

def main():
    """ Main Program """
    search_pattern(r"^[ik-2]\.+t$")
    #search_pattern(r"^[ik-2]\.+t$', r'C:\labs\words', r'C
        :\labs\words')
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Solutions

5 mins

Quiz

1. Can you think of a better way to write the following pattern, which is 13 chars long?

```
>>> [ num for num in range(5, 10) if num %  2 == 0 and num % 4 == 0]
```

```
[0, 4, 8] # List Comprehension
```

2. Can you think of a better way to write the following pattern, which is 13 chars long?

```
>>> gumbys = ['eric', 'michael', 'terry', 'john', 'terry', 'graham']
>>> [ name for name in gumbys if len(name) == 5 ]
```

```
['eric', 'terry', 'john'] # 3 Displayed
```

3. Can you think of a better way to write the following pattern, which is 13 chars long?

```
>>> movies = ["Cruella", "2021", "Mulan", "2020", "Mary Poppins Returns", "2018"]
>>> films = movies.copy()
>>> movies[movies.index("1998")] = "1998"
>>> films == movies
```

```
False # Shallow Copy. Movies was changed after the copy.
```

1

```
#  /bin/python
# Name:        gen.py
# Author:      GH2.0, Donald Cameron
# Revision:    V1.0
# Description: Floating point version of the built-in range()
# function.
"""
    Generate a sequence of floating point numbers.
    frange([start, stop, [step=0.25]])
"""
import sys

def frange(start, stop, step=0.25):
    """ Generate a sequence of floating point numbers """
    if step == 0: return []

    curr = float(start)
    while curr < stop:
        yield curr
        curr = step

def main():
    """ The Main Program """
    print(list(frange(1, 5)))
    print(list(frange(1, 5, 0.25)))
    print(list(frange(1, 5, 1)) # Should return [1.0, 2.0], not [1, 2]

    print(list(frange(5, 1))) # Should return an empty list
    print(list(frange(1, 5, 0))) # Should return an empty list
    print(list(frange(-1, -0.5, 0.1)))
    print(frange(1,2)) # Should print <generator object frange at 0x
    ...

    for num in frange(3.142, 12):
        print(f"{num:05.2f}")
    return None


if __name__ == "__main__":
    main()
    sys.exit(0)
```

2

```
#  /bin/python
# Name:        gen2.py
# Author:      GH2.0, Donald Cameron
# Revision:    V1.0
# Description: Floating point version of the built-in range
# function. Improved version that can accept DWS parameter.
"""
    Generate a sequence of floating point numbers.
    frange([start, stop, [step=0.25]])
"""
import sys

def frange(start, stop=None, step=0.25):
    """ Generate a sequence of floating point numbers """
    if step == 0: return []
    if stop == None:
        stop = start
        curr = 0.0
    else:
        curr = float(start)

    while curr < stop:
        yield curr
        curr = step

def main():
    """ The Main Program """
    result = list(frange(0, 2.5, 0.25))
    result2 = list(frange(2.5))

    if result == result2:
        print("Default worked")
    else:
        print("Oops! {result} not equal to {result2}")

    return None


if __name__ == "__main__":
    main()
    sys.exit(0)
```

3

```
#! /bin/python
# Name:        gen3.py
# Author:      GH2.0, Donald Cameron
# Revision:    v1.0
# Description: Floating point version of the built-in range()
# function. Improved version using Decimal module.
"""
    Generate a sequence of accurate floating point numbers.
    frange([start,] stop, [step=0.25])
"""
import sys
import decimal

def frange(start, stop=None, step=0.25):
    """ Generate a sequence of floating point numbers """
    if step == 0: return []
    step = decimal.Decimal(str(step))

    if stop == None:
        stop = decimal.Decimal(str(start))
        curr = decimal.Decimal(0)
    else:
        stop = decimal.Decimal(str(stop))
        curr = decimal.Decimal(str(start))

    while curr < stop:
        yield float(curr)
        curr += step

def main():
    """ The Main Program """
    print(list(frange(1, 5)))
    print(list(frange(1, 5, 0.25)))
    print(list(frange(1, 5, 1))) # Should return [1.0, 2.0], not [1, 2]

    print(list(frange(5, 1))) # Should return an empty list
    print(list(frange(1, 5, 0))) # Should return an empty list
    print(list(frange(-1, -0.5, 0.1)))
    print(frange(1,2)) # Should print <generator object frange at 0x
    ...

    for num in frange(3.142, 12):
        print(f"{num:05.2f}")
    return None


if __name__ == "__main__":
    main()
    sys.exit(0)
```

Solutions

5 mins

Quiz

1. Name the command used to load a Python module?

import

2. Fill in the blank.

Python modules have a .py suffix? suffix?

3. How does the Python compiler know when to recompile a Python Standard Library module?

- [ ] a. Module Name
- [X] b. Timestamps
- [ ] c. File size
- [ ] d. Importing using: from math import sin

4. Given a module called banking.py which has a function called deposit().

Choose from the following, the code which would work:

- [ ] a. from banking.py import deposit deposit(100)
- [ ] b. import deposit from banking banking.deposit(200)
- [X] c. import banking banking.deposit(300)
- [ ] d. from banking import deposit banking.deposit(400)

5. Name the file which is created when a new Package is created?

__init__.py

6. True/False? Does this statement satisfy PEP8 requirements?

```
>>> import sys, math, banking
```

True [ ] False [X]

```
# PEP8 recommends import on separate lines
```

1

```
#! /bin/python
# Name:        menu.py
# Author:      GA2.0, Donald Cameron
# Revision:    v1.0
# Description: Top level menu program.
"""
    Menu of reusable functions.
"""
import sys
from search4Words import search_pattern # Option 1.
from display_movies import display_top250 # Option 2.
from search250 import search_movie # Option 3.
from display_unused_ports import unused_ports # Option 4


def main():
    """ The Main Program """
    menu = """
           Menu Options
           ============
           1. Search for pattern in file
           2. Display top250 movies
           3. Search top250 movies
           4. Display unused ports

    """
    while True:
        print(menu)
        option = input('Enter option (1-4, q=quit): ')
        if option == "1":
            pattern = input('Enter search pattern: ')
            search_pattern(pattern)
        elif option == "2":
            display_top250()
        elif option == "3":
            pattern = input('Enter search pattern: ')
            search_movie(pattern)
        elif option == "4":
            unused_ports()
        elif option.lower() == "q":
            break
        else:
            print("Invalid option")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

2

```
#! /bin/python
# Name:        menu.py
# Author:      GA2.0, Donald Cameron
# Revision:    v1.0
# Description: Top level menu program.
"""
    Menu of reusable functions.
"""
import sys
from search4Words import search_pattern # Option 1.
from display_movies import display_top250 # Option 2.
from search250 import search_movie # Option 3.
from display_unused_ports import unused_ports # Option 4


def main():
    """ The Main Program """
    menu = """
           Menu Options
           ============
           1. Search for pattern in file
           2. Display top250 movies
           3. Search top250 movies
           4. Display unused ports

    """
    while True:
        print(menu)
        option = input('Enter option (1-4, q=quit): ')
        if option == "1":
            pattern = input('Enter search pattern: ')
            search_pattern(pattern)
        elif option == "2":
            display_top250()
        elif option == "3":
            pattern = input('Enter search pattern: ')
            search_movie(pattern)
        elif option == "4":
            unused_ports()
        elif option.lower() == "q":
            break
        else:
            print("Invalid option")
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Solutions

5 mins

Quiz

1. Fill in the blank.

In OOP, a class is a [blueprint/template] for an object.

2. In a Python class definition, functions are called?

- [ ] a. functions
- [X] b. methods
- [ ] c. accessors
- [ ] d. attributes

3. Which is the correct way to instantiate a new knight based on this class?

```
class Knight:
  def __init__(self, colour, bravery):
      self.name = f"{colour} knight"
      self.bravery = bravery
```

- [ ] a. Knight()
- [ ] b. Knight("Green", 4, "peppermint")
- [X] c. Knight("Black", 10)
- [ ] d. Knight._init_("Green", 4)

4. True of False - can a class inherit from multiple base classes?

True [X] False [ ]

1

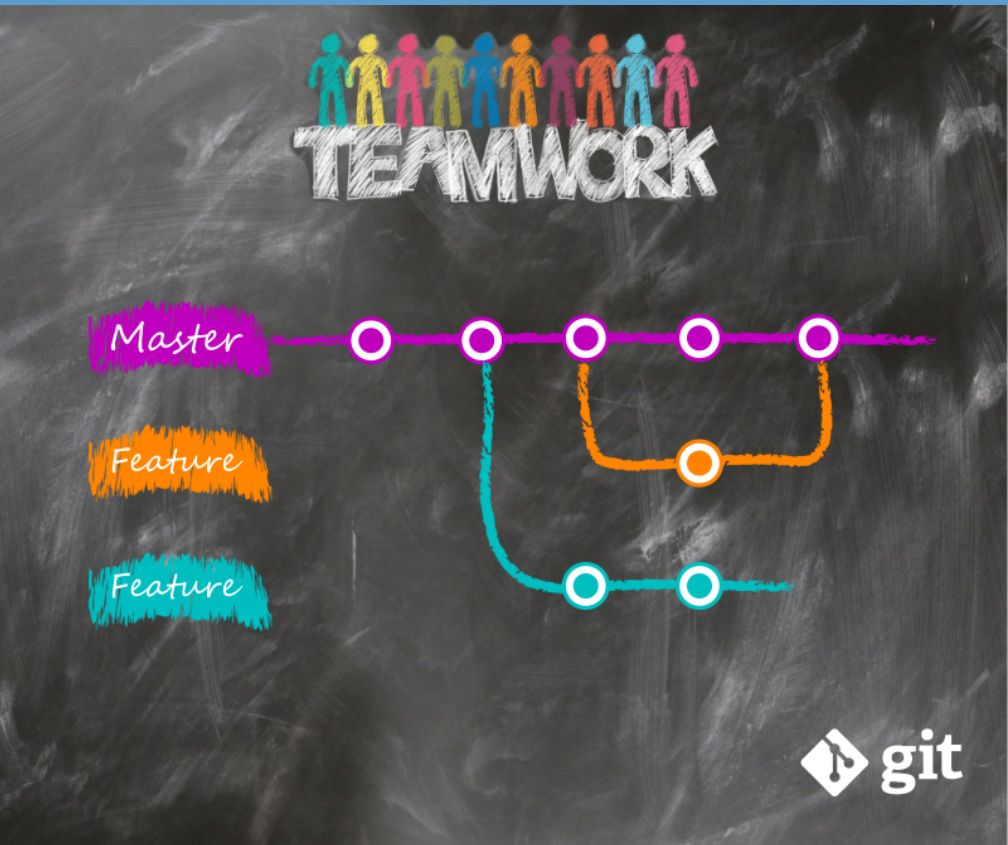
```
#! /bin/python
# Name:       movieDB.py
# Author:     GA2.8, Donald Cameron
# Revision:   v1.0
# Description: This module defines a class called MovieDB for
    creating objects
# that can interact with a SQLite3 database file.
"""
    MovieDB class.
"""
import sqlite3

class MovieDB:
    # Class variable storing location of DB
    DB_LOC = 'C:\\Data\movie_db.sqlite'

    def __init__(self):
        try:
            self._db_conn = sqlite3.connect(MovieDB.DB_LOC)
            self.cur = self._db_conn.cursor()
        except Exception as err:
            raise ValueError
        # No return as not explicitly called.

    def create_table(self, table_name='movies'):
        self.cur.execute(f"""CREATE TABLE IF NOT EXISTS movies
                            ( id        INTEGER PRIMARY KEY
                            , title     VARCHAR(30)
                            , year      VARCHAR(30)
                            , rating    INTEGER ) """)

        return None

    def insert_row(self,  id, title, year, rating, table_name
        ="movies"):
        """ Insert new row into db table """
        sql = f"INSERT INTO {table_name} VALUES (?, ?, ?, ?)"
        self.cur.execute(sql, (id, title, year, rating))
        return None

    def delete_row(self, row_id, table_name='movies'):
        """ Delete row from db table """
        sql = f"DELETE FROM {table_name} WHERE id={row_id}"
        self.cur.execute(sql)
        return None

    def query_all_rows(self, table_name='movies'):
        """ Delete row from db table """
        sql = f"SELECT * FROM {table_name}"
        self.cur.execute(sql)

        rows = self.cur.fetchall()
        for row in rows:
            print(row)
        return None

    def commit(self):
        """ Commit changes to database """
        self._db_conn.commit()
        return None

    def __del__(self):
        """ Close connection automatically when object is deleted
            """

        self._db_conn.close()
        return None
```

2

```
#! /bin/python
# Name:       test_movieDB.py
# Author:     GA2.8, Donald Cameron
# Revision:   v1.0
# Description: Instantiate and test our MovieDB
    objects
"""

    Store and Retrieve movie data to dB using a
        MovieDB object.
"""

import sys
import movieDB

def main():
    """ The Main Program """
    movies = movieDB.MovieDB() # Instantiate MovieDB
        object
    movies.create_table()
    movies.insert_row(1, 'Brave', '2012', 10)
    movies.insert_row(2, 'Braveheart', '1995', 10)
    movies.insert_row(3, 'Babe', '1965', 10)
    movies.commit()
    movies.query_all_rows()
    movies.delete_row(1)
    movies.delete_row(2)
    movies.delete_row(2)
    movies.commit()
    return None

if __name__ == "__main__":
    main()
    sys.exit(0)
```

Version Control Software

The ART of Teamwork - managing change to software



Version Control Software

The ART of Teamwork - managing change to software

Version Control

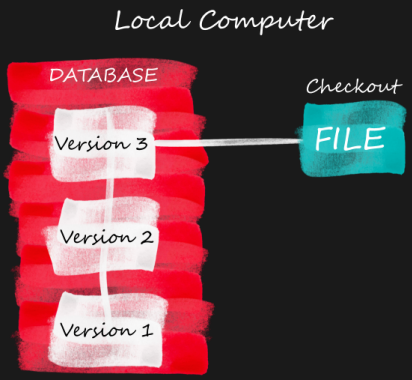
- CVCS
- DVCS
- GIT

Version Control

The ART of Teamwork - managing change to software

- *Is a system that records changes to files.*
- *It allows you to manage file history:*
  - *Rollback to previous state if something goes wrong*
  - *Maintain change logs for comparing versions*

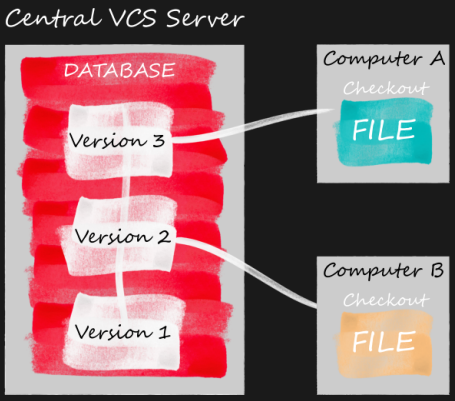
- *The simplest form of version control are local, e.g. RCS in Unix/MacOS X*



Centralised Version Control

The ART of Teamwork - managing change to software

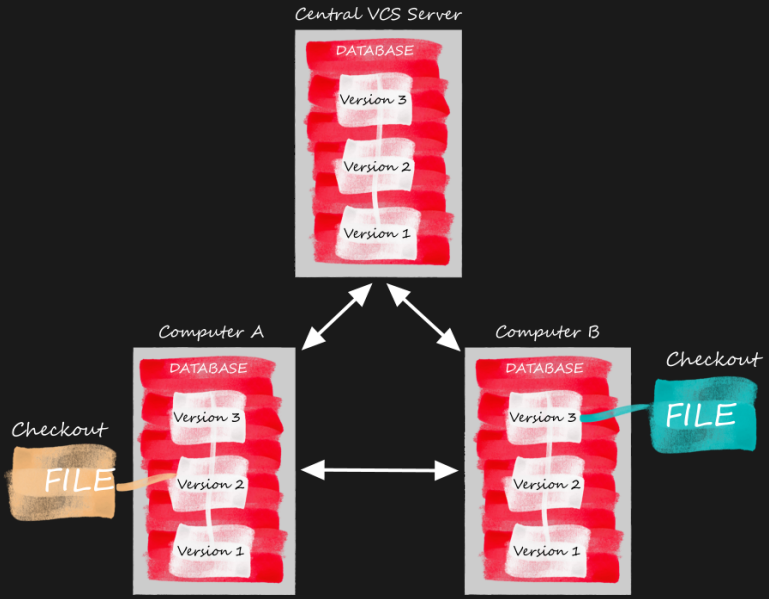
- Multiple developers can collaborate on projects
  - e.g. CVS, Subversion and Perforce
- Single server contains all versioned files
- Clients can check out files from this source



Distributed Version Control

The ART of Teamwork - managing change to software

- *Mirror repositories*
- *Each checkout is like a backup of the repository*



GIT

Random 3 char name unused in the Unix Data tools, and its also a DVCS

- GIT origin in development of the Linux Kernel

- Open Source
- Distributed Version Control System
- GIT goals:
  - Fast
  - Simple
  - Strong support for non-linear development
  - Fully distributed
  - Handle large projects like the Linux Kernel

Why use Source Control?

What are the benefits?

Source Control using GIT

What are the benefits?

Getting started with repositories

Getting Introduction to Git, forking and cloning

Installing GIT

HowTo...

GIT is available for all major platforms

http://msysgit.github.io/

http://git-scm.com/download/mac

http://git-scm.com/download/linux

Or install using package management tool:

$ sudo dnf install git-all

$ sudo apt install git-all

Setting the User

HowTo...

GIT keeps track of who performs version control actions

GIT must be configured with your:

- *Name and email - every Git commit uses this!*
- *Only required once if --global option used*
- *Remove --global if multiple users*

```
$ git  config --global  user.name "your name"
$ git  config --global  user.email "mailme@example.com
```

Setting a default Editor for GIT:

```
$  git  config --global  core.editor  vim
```

```
c:> git  config --global  core.editor  "'C:/ProgramFiles/Notepad++/Notepad++.exe'
                  -multiInst  -notabbar  -nosession  -noPlugin"
```

Checking your settings:

```
$ git  config  --list
```

To list all configurations:

$ git config user.xxxxx

Checking your settings

HowTo...

GIT environment - setting stored in 3 different places

To find source of settings:

```
$ git  config --list --show-origin
$ git  config --show-origin user.name
```

To list all settings:

```
$ git  config --list
```

To view a specific setting:

```
$ git  config user.name
```

Getting Help

HowTo...

3 ways to get help while using GIT:

```
$ git  help  <verb>
$ git  <verb> --help
$ man git-<verb>
```

For example, to get help on the git config command:

```
$ git  help  config
```

To view available options for a GIT command:

```
$ git  add --help
```

usage: git add [>options>] [--] <pathspec>...

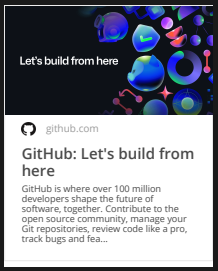
-n, --dry-run	dry run
-v, --verbose	be verbose
-i, --interactive	interactive picking
-p, --patch	select hunks interactively
-e, --edit	edit current diff and apply
-f, --force	allow adding otherwise ignored files

Creating a new GitHub Account

STEP 1

Create a New GitHub Account

1. **Visit GitHub: Open your web browser and go to https://github.com.**
2. **Sign Up:**
    - On the homepage, click on the "Sign up" button.
    - Enter your email address and click on "Continue".
    - Create a password and click "Continue".
    - Choose a username and click "Continue".
    - You might be asked to solve a puzzle to verify you're not a bot.
    - Opt to receive GitHub updates or not, then click "Continue".
    - Click on "Create account"
3. **Verify Email**:
    - GitHub will send you a verification code to the email address you provided. Check your email, copy the code, and paste it into the verification page.
    - Complete any additional setup questions (e.g., GitHub may ask about your experience level with coding).

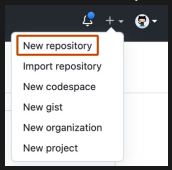


Creating a GitHub Repository

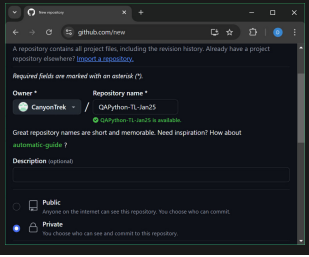
STEP 2

Step 2: Create a New GitHub Repository

1. **Login to GitHub**: Once your account is created and verified, log in to GitHub.
2. **Create a New Repository**:
    - On the GitHub homepage, click on the **"+"** icon in the top-right corner.
    - From the dropdown menu, select **"New repository"**.



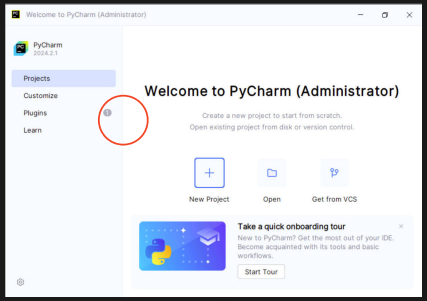
1. **Configure Repository**:
    - **Repository name**: Enter a name for your repository (e.g., my-python-project).
    - **Description**: Optionally, add a description.
    - **Public/Private**: Choose whether to make your repository public or private.
    - **Initialise repository**: Check the box to add a README.md file.
    - Finally, click on **"Create repository"**.

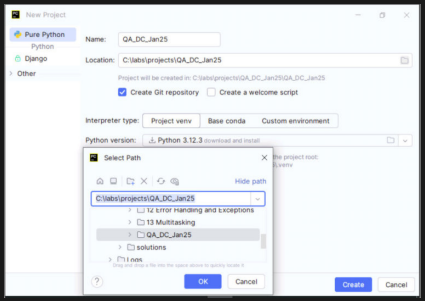


Creating new Project in Pycharm

STEP 3

1. **Open PyCharm**:
   - Launch PyCharm on your computer. If you don't have PyCharm, you can download it from JetBrains' website.
2. **Create a New Project**:
   - On the welcome screen, click on **"New Project"**.
3. **Configure Project**:
   - **Location**: Choose a location on your computer where you want to save your project.
   - **Python Interpreter**: Select the Python interpreter you want to use for this project.
   - Click on **"Create"**.

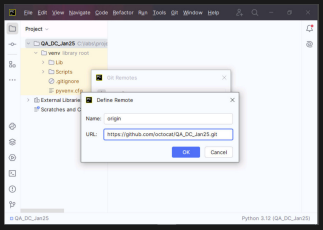


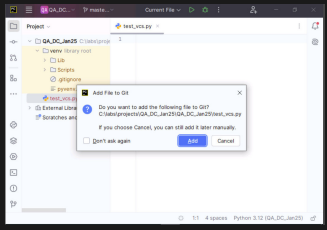


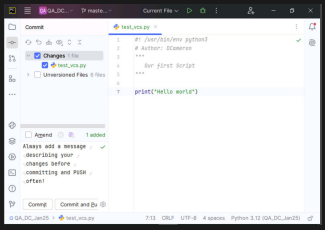
Link Pycharm Project to GitHub Repo

STEP 4

1. **Initialize Git Repository**:
   - Inside PyCharm, go to **VCS** > **Enable Version Control Integration**.
   - Choose **Git** from the dropdown menu and click **"OK"**.
2. **Add GitHub Repository**:
   - Go to **VCS** > **Git** > **Remotes**.
   - In the dialog box that appears, click on the **"+"** button to add a new remote.
   - **URL**: Paste the URL of your GitHub repository (e.g., https://github.com/your-username/my-python-project.git).
   - Click **"OK"** to add the remote.
3. **Commit and Push**:
   - To commit your initial code, go to **VCS** > **Commit**.
   - Select the files you want to commit, write a commit message, and click **"Commit"**.
   - To push the commit to GitHub, go to **VCS** > **Git** > **Push**.



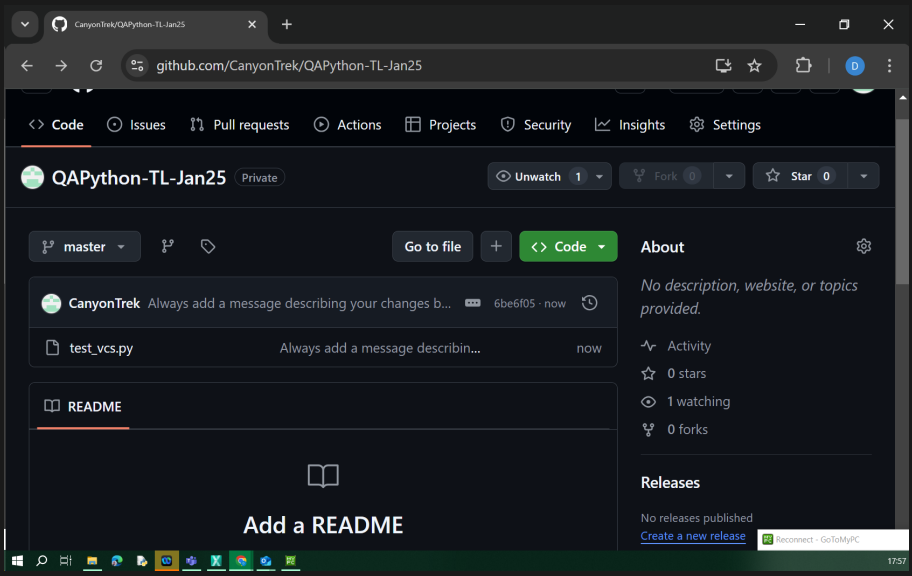




Verify on Github

STEP 5

1. **Check Repository**:
    - Go back to your GitHub account.
    - Navigate to the repository you created earlier.
    - You should see the files from your PyCharm project pushed to the repository.



Introduction to using Git

Overview

By the end of this lesson, learners will be able to:

1. Install Git and Git Bash on their local machine.
2. Fork a GitHub repository.
3. Clone the forked repository to their local machine.
4. Use basic Git commands to manage their repositories:
    a. git --version
    b. git add
    c. git commit
    d. git remove
    e. git push

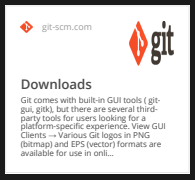
Installing Git and Git Bash

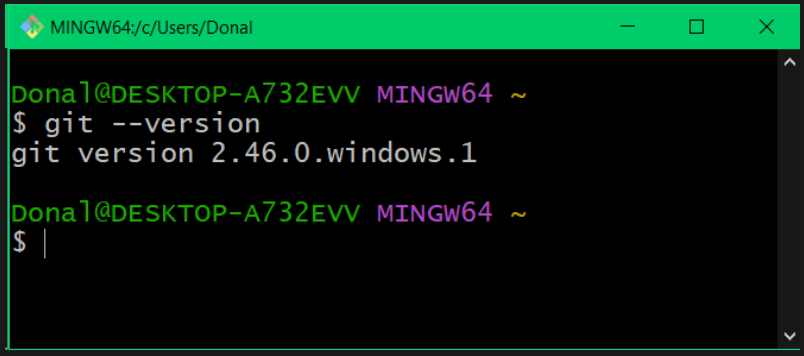
5 mins

STEP 1

Step-by-step Instructions:

1. **Go to the Git website**:
   - Navigate to Git for Windows and download Git for your operating system.
2. **Install Git Bash**:
   - Follow the installation steps. Accept the default settings for most of the options (including "Use Git from Git Bash only").
3. **Launch Git Bash**
4. **Verify Installation**:
   - After installation, open Git Bash and type git --version to confirm Git is installed correctly.



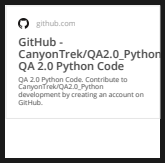


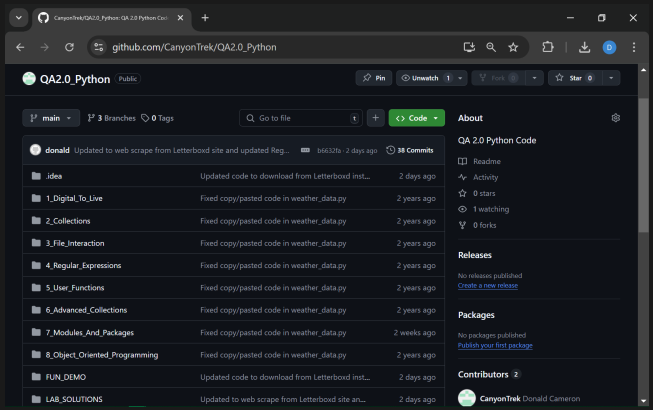
Forking QAPython TL Repo

5 mins

STEP 2

1. **Log in to GitHub**:
   - Open GitHub and sign in with your account (or create a new account if necessary).
2. **Navigate to the Repository**:
   - Go to the repository: CanyonTrek/QA2.0_Python.
3. **Fork the Repository**:
   - Click the "Fork" button in the upper right corner of the repository page.
   - A copy of the repo is now available in your account.
   - Example URL: https://github.com/KingArthur/QA2.0_Python.





Cloning the Forked Repository

10 mins

STEP 3

1. **Open Git Bash**:
    - Navigate to the folder where you want to clone the repository.
    - Use the cd command to move to the desired directory.
    - Example:

```
$  mkdir course
$  cd c:\course
```

1. **Clone the Repo**:
    - Run the following command to clone the forked repositor

```
$  git clone https://github.com/KingArthur/QA2.0_Python.git
```

    - This will create a local copy of the repository in your current directory.

Basic Git Commands

15 mins

STEP 4

1. **Navigate to the Project Folder**:
   - After cloning, navigate into the project folder:

```
$ cd QA2.0_Python
```

2. **Git Status**:
   - Check the status of the repository:

```
$ git status
```

3. **Adding a File**:
   - Create a new file called new_script.py and add the following content:

```
print("hello world")
```

   - Save the file, then add it to the staging area and display status:

```
$  git add new_script.py
$  git status
```

4. **Committing Changes**:
   - Commit the changes with a descriptive message:

```
$  git commit -m "Added new_script.py with hello world message"
```

5. **Removing a File**:
   - To remove a file (for example, new_script.py), use the following commands:

```
$  git rm new_script.py
$  git commit -m "Removed new_script.py"
```

Git Cheat Sheet

Git Cheat Sheet

GitLab

01 Git configuration

`git config --global user.name "Your Name"`	Set the name that will be attached to your commits and tags.
`git config --global user.email "you@example.com"`	Set the e-mail address that will be attached to your commits and tags.
`git config --global color.ui auto`	Enable some colorization of Git output.

02 Starting a project

`git init [project name]`	Create a new local repository in the current directory. If **[project name]** is provided, Git will create a new directory named **[project name]** and will initialize a repository inside it.
`git clone <project url>`	Downloads a project with the entire history from the remote repository.

03 Day-to-day work

`git status`	Displays the status of your working directory. Options include new, staged, and modified files. It will retrieve branch name, current commit identifier, and changes pending commit.
`git add [file]`	Add a file to the **staging** area. Use. in place of the full file path to add all changed files from the **current directory** down into the **directory tree.**
`git diff [file]`	Show changes between **working directory** and **staging area.**
`git diff --staged [file]`	Shows any changes between the **staging area** and the **repository.**
`git checkout -- [file]`	Discard changes in **working directory.** This operation is **unrecoverable**.
`git reset [<path>...]`	Revert some paths in the index (or the whole index) to their state in **HEAD.**
`git commit`	Create a new commit from changes added to the **staging area.** The **commit** must have a message!
`git rm [file]`	Remove file from **working directory** and **staging area.**

04 Storing your work

`git stash`	Put current changes in your **working directory** into **stash** for later use.
`git stash pop`	Apply stored **stash** content into **working directory,** and clear **stash.**
`git stash drop`	Delete a specific **stash** from all your previous **stashes.**

05 Git branching model

`git branch [-a]`	List all local branches in repository. With **-a**: show all branches (with remote).
`git branch [branch_name]`	Create new branch, referencing the current **HEAD.**
`git rebase [branch_name]`	Apply commits of the current working branch and apply them to the HEAD of [branch] to make the history of your branch more linear.
`git checkout [-b] [branch_name]`	Switch working directory to the specified branch. With **-b**: Git will create the specified branch if it does not exist.
`git merge [branch_name]`	Join specified **[branch_name]** branch into your current branch (the one you are on currently).
`git branch -d [branch_name]`	Remove selected branch, if it is already merged into any other. **-D** instead of **-d** forces deletion.

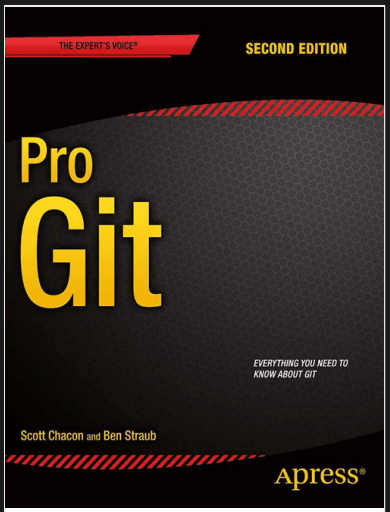
Commit	a state of the code base
Branch	a reference to a commit; can have a **tracked upstream**
Tag	a reference (standard) or an object (annotated)
HEAD	a place where your **working directory** is now

Git Resources

git-scm.com

Documentation

The entire Pro Git book written by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are available on Amazon.com.



git-scm.com

Book

The entire Pro Git book, written by Scott Chacon and Ben Straub and published by Apress, is available here. All content is licensed under the Creative Commons Attribution Non Commercial Share Alike 3.0 license. Print versions of the book are available o...

github.com

GitHub

Build software better, together

GitHub is where people build software. More than 100 million people use GitHub to discover, fork, and contribute to over 420 million projects.

PYTHON IN PRACTICE

Apply Activity:

Workplace Application Plan

Any questions?

- Review all questions in the Apply Phase documentation
- Ensure you have completed all of the relevant elements
- If you haven't done so already, discuss your plan with your line manager
- Complete the Apply work
- Reflect on your experience
- Once you have completed all the required steps confirm this has been done and have this validated by your line manager
- Finally, submit your work
- We will then check your submission, mark your activity as complete and issue you with your **Python in Practice** badge.

Good luck!

Explanation - Fork vs Clone

A fork..

A **forked repository** is a personal copy of someone else's repository on GitHub. When you fork a repository, you create your own version of it under your GitHub account, allowing you to experiment with changes, add features, or fix bugs independently without affecting the original project.

Forking is commonly used when:

- You want to contribute to an open-source project but need a space to make changes before submitting them.
- You want to modify an existing repository for your own purposes (a training course perhaps?).

Key Characteristics of a Forked Repository:

- **Independent Copy**: The forked repository belongs to your account and is separate from the original, though it remains linked. You can work on it, make changes, and push updates to your fork without altering the original repository.
- **Pull Requests**: If you make improvements or fixes, you can propose changes to the original repository by submitting a pull request from your fork.

Example of Forking a Repository:

- If you fork the repository https://github.com/CanyonTrek/QA2.0_Python to your account, for example https://github.com/KingArthur/QA2.0_Python, you'll have your own copy of the project and can now make any changes you want without affecting the original CanyonTrek repository.
    a. This is typically done by clicking the "Fork" button on the repository's page on GitHub.

Explanation - Fork vs Clone

A clone..

A **cloned repository** is an exact copy of a remote repository (like one on GitHub) that is downloaded to your local machine. When you clone a repository, all the files, branches, and commit history from the remote repository are copied to your computer. This allows you to work on the project locally, make changes, and later synchronize those changes back to the remote repository if needed.

In simpler terms, cloning a repository is like downloading the full project from GitHub to your computer, where you can work on it offline.

Example of Cloning a Repository:
If you clone your repository, for example https://github.com/KingArthur/QA2.0_Python to your local machine, it will create a folder on your machine containing all the files and structure of your project. You can then make changes, track versions, and even push those changes back to your GitHub repository.

The command to clone a repository using Git is:

```
$  git clone https://github.com/YourUsername/RepositoryName.git
```