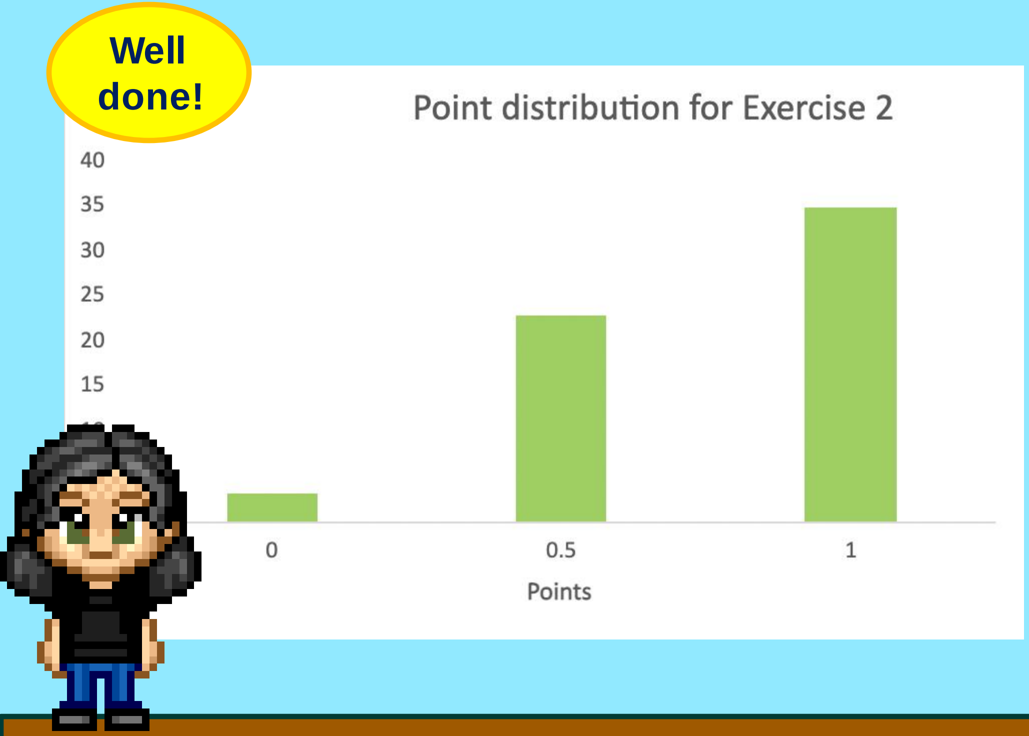
PCL2-Tutorial 06

Lucas Suomela, Rong Li, Tosca Peruzzi-Vieli, Isabelle Cretton

Welcome!

Exercise 2: feedback

Well done!



Grading was rigorous based on the following criteria: **correct task implementation and topic understanding**, code readability, best coding practices, effort, and documentation (including comments).

Given the exercise's solvability by ChatGPT, grading leniency was minimal, with a strong emphasis on rewarding effort!

Some comments:

- 21 people did not release the assignment to us!
- Average points per person: 0.76! **Well done.**

References

Think back to immutable vs. mutable data

- A variable name references to where the data is located in memory
- If the underlying object is mutable, then any modifications done will persist
- If the underlying object is immutable, modifications will not persist.

How References Work

- Assignment of objects to variables creates a reference, not a copy.
- Multiple variables can reference the same object.

```
# Assigning a list to a variable
original_list = [1, 2, 3]
# Creating a reference to the same list
reference_list = original_list

# Modifying the original list
original_list.append(4) # this will also change the reference
```

```
print(original_list)
# output:  [1, 2, 3, 4]
```

```
print(reference_list)
# output:  [1, 2, 3, 4]
```

Good article here

READ ME!!

Mutable vs Immutable

```python
# Mutable object example
print("Mutable object example:")
list1 = [1, 2, 3]
list2 = list1
list2.append(4)

print("list1:", list1)  # Output will show list1 is affected by changes in list2
print("list2:", list2)  # list2 is a reference to the same list as list1

# Immutable object example
print("Immutable object example:")
int1 = 10
int2 = int1
int2 += 5

print("int1:", int1)  # Output will show int1 remains unchanged
print("int2:", int2)  # int2 is a separate copy after the operation
```

References and Functions

```python
def modify_data(data_dict):
    # Modifying the dictionary inside the function
    data_dict['age'] = 30
    data_dict['new_key'] = 'new_value'
    return data_dict

# Dictionary to be passed to the function
original_dict = {'name': 'John', 'age': 25}

# Passing the dictionary to the function
modified_dict = modify_data(original_dict)

print("Original Dictionary after function call:", original_dict)
# Shows original_dict is modified because it is passed by reference

print("Modified Dictionary:", modified_dict)
# Shows both variables reflect the changes made inside the function
```

These examples highlight how references work in Python and the impact of mutable and immutable types on variable behavior, especially in the context of function calls and data modifications.

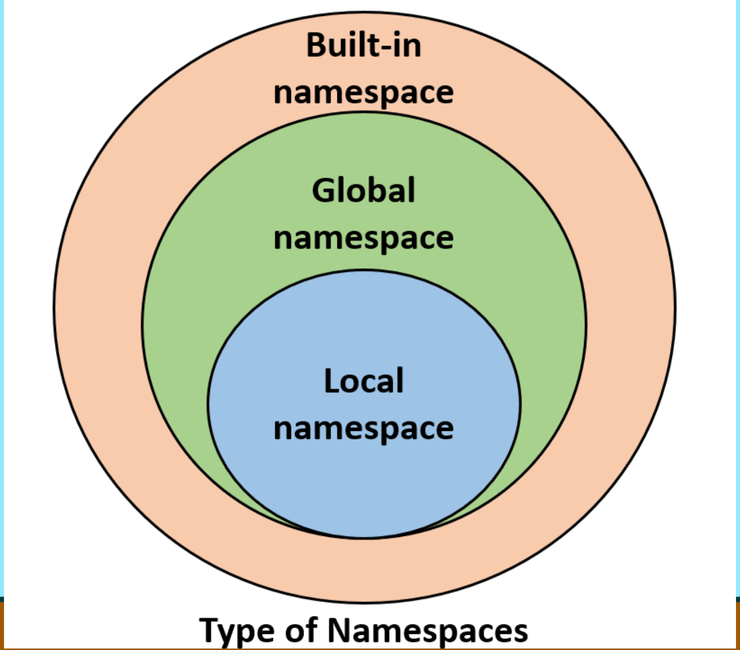
Namespaces

- A namespace is a container (dictionary) where names are mapped to objects, such as variables and functions
- Enables the Python interpreter to distinguish between identifiers with the same name but in different namespaces
- Scope Resolution: The Process of accessing variables across different namespaces
- Follows LEGB rule: Local -> Enclosed -> Global -> Built-in

- Variables in Python are names and belong to exactly one namespace
- Modify namespaces dynamically by adding, changing or removing names

Modules, classes, functions and methods have their own local namespace.

Namespaces: Types of Namespaces



Built-in

Contains built-in functions and exceptions, available when Python interpreter starts. Exists until the interpreter quits

Global

Contains names from the current module or script, exists until the script ends or the interpreter is closed

Local

Contains names defined within a function or block, accessible only inside that function or block. Created when a function is called and deleted when the function returns or finishes execution

Namespaces: Keywords

tldr; `global` is used to modify variables in the global scope from within a function, while `nonlocal` is used to modify variables in the nearest enclosing (non-global) scope, particularly in nested functions.

`nonlocal`

- Can be used to modify variables from the enclosing namespace.
- The `nonlocal` keyword is used in nested functions to refer to variables in the nearest enclosing scope that is not global.
- It allows a nested function to modify a variable in its parent function's scope.
- `nonlocal` is useful for working with closures and function factories, where you need to modify variables from an outer, but not global, scope.

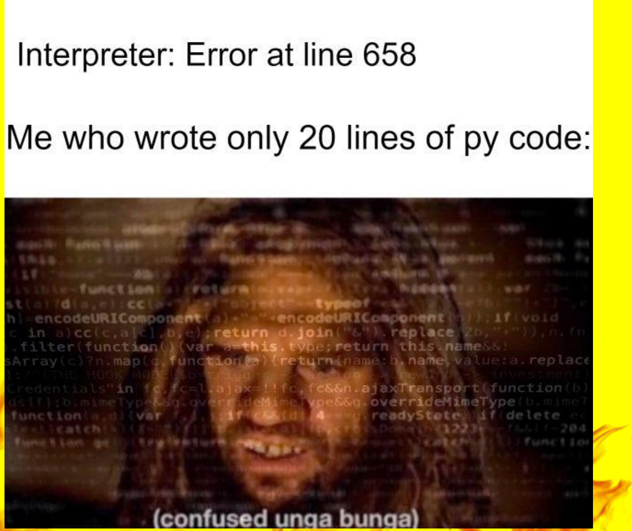
```
def outer():
   x = 5
   def inner():
      nonlocal x  # Reference x from the nearest enclosing scope
      x = 10
   inner()
   print(x)  # Outputs 10, as inner() modifies the outer's x
outer()
```

`global`

- Used to modify global variables from within a non-global scope. Accessing global variables for reading is possible without `global`
- The `global` keyword is used to declare that a variable inside a function refers to a globally scoped variable.
- It allows a function to modify a variable that is defined outside the function's scope.
- Using `global`, a function can change the value of a variable defined at the top level of the script or module.

```
x = 10

def modify_global():
   global x  # Declare that x is the global variable
   x = 20

modify_global()
print(x)  # Will print 20, as the global x has been modified
```

ERRORS


Interpreter: Error at line 658
Me who wrote only 20 lines of py code:
(confused unga bunga)

We've all dealt with them.... how can we handle them?

Handling Errors with Exceptions

Exception handling in programming is crucial for managing errors gracefully and ensuring the program doesn't crash unexpectedly.

```
except {error_type}:
```

```
try:
```

```
try:
        if not isinstance(text, str):
raise TypeError("Input must be a string")
```

```
except TypeError as e:
        print("Error:", e)
```

```
except IOError:
        print("Error: An error occurred while reading the file.")
```

```
except FileNotFoundError:
     print("Error: The specified file was not found.")
```

Exceptions are special objects or states in a program used to manage errors or unexpected events

Can you think of any instances in your own programming experience where you might have needed these?

some typical use cases

Reading and processing files with inconsistent data formats

- **Scenario**: A program processes a file where most lines follow a specific format, but some lines are malformed or don't match the expected pattern.
- **Exception use**: Catch and handle parsing errors for the problematic lines, possibly logging the error and skipping to the next line without stopping the entire process.

User input validation

- **Scenario**: A user inputs data into a form or interface that expects data in a certain format or within certain limits, but the input is invalid.
- **Exception use**: Catch validation errors and prompt the user to re-enter the data correctly, without crashing the program.

Handling text files with unknown or mixed encodings

- **Scenario**: A program reads text files that are assumed to be in UTF-8 encoding, but some files are actually in a different encoding, leading to decoding errors.
- **Exception use**: Catch the **UnicodeDecodeError** to handle files with unexpected encodings. The program can then try different encodings, log the issue, or notify the user to correct the file format.

And there are many more...

(**Here**'s a great .md file with lots of examples)

How to use them in your code

```
def read_text_file(file_path, default_encoding='utf-8', fallback_encoding='iso-8859-1'):
    try:
        with open(file_path, encoding=default_encoding) as file:
            print("Reading file with default encoding (UTF-8)...")
            content = file.read()
            print("File read successfully with UTF-8 encoding!")
            return content
    except UnicodeDecodeError:
        print("Failed to read file with UTF-8 encoding. Trying fallback encoding...")
        try:
            with open(file_path, encoding=fallback_encoding) as file:
                content = file.read()
                print(f"File read successfully with {fallback_encoding} encoding!")
                return content
        except UnicodeDecodeError:
            print("Failed to read file with fallback encoding as well.")
        except Exception as e:
            print(f"An unexpected error occurred: {e}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

What's going on in this function?

Arguments

The arguments that are given after the name of the program in the command line shell of the operating system are known as **Command Line Arguments**.

"Command line arguments are a powerful way to pass parameters to a program when it is executed. They allow users to customize the behavior of the program without modifying its source code. Command-line arguments are used extensively in the Unix/Linux command-line interface and are also commonly used in scripting languages like Python."

(Link to article here)

Real-world examples of Python scripts using command line arguments include:

- A script for converting a CSV file to JSON format, taking the CSV file as input and allowing optional arguments for output format, delimiter, and column data types.
- A script for machine learning, which processes a dataset file with options to specify the machine learning algorithm, hyperparameters, and output format.
- A script that processes natural language text, which might take a text file as input, with optional arguments to specify the language model to use, the analysis type (like sentiment analysis or entity recognition), and the output format for the results.

```
parser = argparse.ArgumentParser(description='Simple Sentiment Analysis Script')
parser.add_argument('text', type=str, help='Text to analyze sentiment for')
parser.add_argument('--output_format', choices=['text', 'json'], default='text', help='Output format of the sentiment analysis result')
args = parser.parse_args()
```

Arguments: keyword vs. positional

Positional Arguments

- These are arguments that need to be passed in the correct order.
- The function call must match the order of parameters in the function definition.

```
def display_info(name, age):
    print(f"Name: {name}, Age: {age}")

display_info("Alice", 30)
```

Keyword Arguments

- These are arguments specified by the name of the parameter, regardless of the order.
- They are often used for optional parameters and enhance readability.

```
def display_info(name, age):
    print(f"Name: {name}, Age: {age}")

display_info(age=30, name="Alice")
```

Basically:

- **Keyword arguments = dictionary**
- **Positional arguments = tuple**

A bit of repetition of arguments as you've seen them before

Arguments: keyword vs. positional for CLI

In command line interfaces (CLI), **keyword arguments** (options or flags) are marked by a preceding hyphen for single-letter options (-) or two hyphens (--) for multi-word options.

Positional Arguments

- These are required arguments that must be provided in the specific order they are defined in the CLI tool.
- They are typically used for essential inputs that the command needs to function.

Keyword Arguments (Options or Flags)

- These arguments are optional and identified by a name, usually preceded by one or two dashes (e.g., **-v** or **--verbose**).
- They modify the behaviour of the command and can be placed in any order in the command line.

```
import argparse

def main():
    # Initialize the argument parser
    parser = argparse.ArgumentParser(description="Greet a user with a custom message")

    # Define positional argument for the user's name
    parser.add_argument('name', type=str, help="Name of the user to greet")

    # Define optional argument for the greeting message (keyword arg)
    parser.add_argument('--greeting', type=str, default="Hello",
                        help="Custom greeting message (default: 'Hello')")

    # Parse arguments
    args = parser.parse_args()

    # Output the greeting
    print(f"{args.greeting}, {args.name}!")

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

```
# Using only the positional argument
python greeting.py Alice

# Using both positional and keyword arguments
python greeting.py Alice --greeting "Welcome"
```

To run this script in the terminal

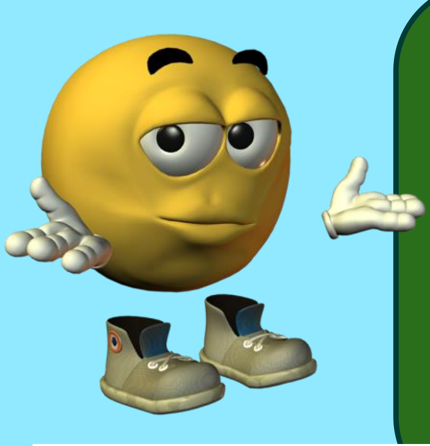
Command Line Developer Tools
Command Line Tools (OS X 10.9).pkg

argparse

- **argparse** is a Python module for parsing command-line arguments, included in the standard library.
- It simplifies creating user-friendly command-line interfaces by abstracting argument parsing complexity.
- Automatically generates helpful usage and help messages, guiding users on how to run the script.
- Supports complex parsing scenarios, including optional and positional arguments, variable argument lists, and sub-commands.
- Enhances the usability and maintainability of command-line applications.

```python
import argparse
import nltk
from nltk.tokenize import word_tokenize
from nltk.probability import FreqDist

def process_text(text):
    # Tokenize the text
    tokens = word_tokenize(text)
    # Calculate frequency distribution
    freq_dist = FreqDist(tokens)
    return freq_dist

def main():
    # Initialize parser
    parser = argparse.ArgumentParser(description='NLP  Script: Text Tokenization and Frequency Distribution')
    # Adding argument
    parser.add_argument('text',  type=str, help='Text to process')
    # Parse arguments
    args = parser.parse_args()

    # Process the text
    freq_dist = process_text(args.text)

    # Print the frequency distribution of the text
    for word, frequency in freq_dist.most_common():
        print(f'{word}:  {frequency}")

if __name__ == '__main__':
    main()
```

Exercise 06



- Create a command-line interface using the `argparse` library
- Adapt a given implementation for calculating the Levenshtein distance, allowing for variable weights (Make sure you fully understand the function!)
- Use proper error handling
- What do you prefer, OOP or Functional Programming? Choose whatever approach you like more!

When you open VS Code using code .
You know, I'm something of a CLI user myself