**Week 3 Python**

**Data structure Day 1**

* .Function Definition:
* `fib(n)` is defined to print the Fibonacci series up to the given boundary `n`.
* `fib2(n)` is defined to return a list containing the Fibonacci series up to `n`.
* Fibonacci Calculation:
* The Fibonacci series is generated using two variables, `a` and `b`, initialized to 0 and 1.
* The series is calculated in a loop using the formula `a, b = b, a + b`.
* Printing vs. Returning:
  + `fib(n)` prints the series while `fib2(n)` returns it as a list.
* Function Calls:
  + `fib(2000)` and `fib(0)` are examples of function calls.
  + `fib2(100)` returns the Fibonacci series up to 100 and stores it in the variable `f100`.
* Return Statement:
  + The `return` statement is used to return a value from a function. It returns `None` if omitted.
* List Manipulation:
  + `result.append(a)` adds the value of `a` to the `result` list.
* Method Calls:
  + `result.append(a)` demonstrates calling a method (`append()`) on a list object (`result`).
* Function Renaming:
  + Functions can be assigned to other names, like `f = fib`, allowing you to call them using the new name.
* None:
  + Functions without a `return` statement return `None`. It's Python's way of representing no value.
* Efficiency:
  + The `result.append(a)` method is more efficient than `result = result + [a]` for building lists.

**More on lists:**

* `list.append(x)`: Add an item to the end of the list.
* `list.extend(iterable)`: Extend the list by appending items from an iterable.
* `list.insert(i, x)`: Insert an item at a given position.
* `list.remove(x)`: Remove the first occurrence of an item with a specific value.
* `list.pop([i])`: Remove and return an item at a given position or the last item if no index is specified.
* `list.clear()`: Remove all items from the list.
* `list.index(x)`: Find the index of the first item with a specific value.
* `list.count(x)`: Count the number of times a specific value appears in the list.
* `list.sort()`: Sort the list in place.
* `list.reverse()`: Reverse the list in place.
* `list.copy()`: Create a shallow copy of the list.

These methods allow you to manipulate and work with lists efficiently.

**Using List as stack and queues:**

You can use a list as a stack for "last-in, first-out" operations by using `append()` to add items to the top and `pop()` to retrieve items from the top.

For a queue with "first-in, first-out" operations, it's better to use `collections.deque` for efficiency. It provides fast appends and pops from both ends.

**Stack using list:**

**A black screen with white text

Description automatically generated**

**Queue using collections.deque:**

A black screen with white text

Description automatically generated

Lists are not efficient for queues because inserting or popping from the beginning of a list is slow due to shifting elements. Use `collections.deque` for faster queue operations.

**List comprehensions:**

List comprehension provides a concise way to create lists in Python. They consist of an expression followed by a for clause and optional if.

A black rectangle with colorful text

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A black rectangle with colorful text

Description automatically generated

A black and white screen with text

Description automatically generated

A screen shot of a computer program

Description automatically generated

**Nested List Comprehensions:**

You can use list comprehensions for complex tasks, like transposing a matrix, but prefer built-in functions when possible. For transposing, use zip():

A screenshot of a computer code

Description automatically generated

**The del statement / Tuples and Sequences:**

In Python, you can use the del statement to remove items from a list by specifying their index or clear the entire list. For example:

A screenshot of a computer

Description automatically generated

* Tuples are similar to lists but are immutable, meaning you can't change their elements after creation. They are often used for different purposes.
* Tuples are defined with parentheses, and they can be nested. You can use sequence unpacking to assign values from a tuple to variables:

A screenshot of a computer

Description automatically generated

* Tuples with one item require a trailing comma to distinguish them from parentheses, like this:

A black and white screen with white text

Description automatically generated

* Tuples are useful for situations where you want to ensure data remains unchanged, and sequence unpacking simplifies variable assignment from tuples.

**Sets:**

* Python has a set data type that stores an unordered collection of unique elements.
* You can create a set using curly braces or the set () function.
* Sets are handy for membership testing and mathematical operations like union, intersection, difference, and symmetric difference.

A black rectangular object with green text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A black and white screen with text

Description automatically generated

**Dictionaries:**

* Dictionaries are another data type in Python, used for key-value pairs.
* Dictionaries are created with curly braces or the dict() constructor.
* Keys must be unique, and you can store, extract, or delete values based on keys. Here's a quick overview.

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Looping Techniques:**

You can loop through dictionaries using the items () method to get both the key and value simultaneously:

A screen shot of a computer

Description automatically generated

You can loop through sequences and get the index and value using enumerate():

A black screen with white text

Description automatically generated

To loop over multiple sequences in parallel, use zip():

A screenshot of a computer screen

Description automatically generated

For looping in reverse, use reversed():

A screen shot of a computer

Description automatically generated

To loop over a sequence in sorted order, use sorted():

A computer screen with text on it

Description automatically generated

Avoid modifying a list while looping over it, and consider creating a new list instead. For example, you can filter out NaN values from a list:

A computer screen with text

Description automatically generated

**Comparing Sequences and Other Types:**

* Sequence objects can be compared to other objects of the same type using lexicographical ordering. This means they are compared element by element, with the first differing element determining the outcome.
* If all elements are equal, the sequences are considered equal.

Examples of comparisons:

* + (1, 2, 3) < (1, 2, 4)
  + [1, 2, 3] < [1, 2, 4]
  + 'ABC' < 'C' < 'Pascal' < 'Python'
  + (1, 2, 3, 4) < (1, 2, 4)
  + (1, 2) < (1, 2, -1)
  + (1, 2, 3) == (1.0, 2.0, 3.0)
  + (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)

You can compare objects of different types if they have appropriate comparison methods. Otherwise, it raises a TypeError exception.

**Day 2:**

**How to create list.**

A Python list is like a container that can hold multiple items (numbers, words, etc.). To create a list, you put your items inside square brackets [ ], separated by commas.

A screenshot of a computer

Description automatically generated

We can also have nested list, which we can have list as items:

A computer screen shot of a computer

Description automatically generated

**Access list elements:**

* In Python, you can access elements in a list using the index operator [].
* Lists start with an index of 0, so the first item is at index 0, the second at 1, and so on. For example
* We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4.
* Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.
* Nested lists are accessed using nested indexing.

A screen shot of a computer program

Description automatically generated

Output:

A black background with white letters

Description automatically generated

**Negative indexing:**

In Python, you can use negative indexing to access items in a list or sequence.

* Index -1 refers to the last item.
* Index -2 refers to the second-to-last item.
* And so on, with each negative index counting backward from the end of the list.

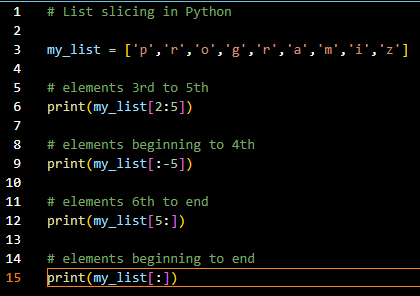
A computer screen with text

Description automatically generated

Output:



**How to slice lists in Python?**



**Output:**

A black screen with white text

Description automatically generated

**Add/Change List Elements:**

* **Lists are mutable, meaning their elements can be changed unlike stings or tuple.**
* **We can use the assignment operator = to change an item or range of items.**

A screenshot of a computer program

Description automatically generated

**Output.**

A screenshot of a computer

Description automatically generated

* **We can add an item on the list using append ().**
* **Add several items using extend().**

A screenshot of a computer program

Description automatically generated

* **Use the + operator to combine two lists.**
* **This is called concatenation.**
* **The \* operator repeats a list for the given number of times.**

A screenshot of a computer program

Description automatically generated

**Delete/Remove List Elements**

* In Python, you can delete or remove elements from a list using various methods.
* The two primary methods for removing elements from a list are using the `del` statement and the `remove()` method.

Here's a summary of each method along with an example:

* Using the `del` Statement:
  + The `del` statement allows you to remove an element from a list by specifying its index.
  + It can also be used to delete entire slices of a list.
  + This method directly modifies the original list.

Example:

* ```python
* # Create a list
* my\_list = [1, 2, 3, 4, 5]
* # Remove the element at index 2 (which is '3')
* del my\_list[2]
* print(my\_list) # Output: [1, 2, 4, 5]

```

* Using the `remove()` Method:
  + The `remove()` method is used to delete the first occurrence of a specific value from the list.
  + It does not require specifying an index but rather the actual value to be removed.
  + If the value appears multiple times in the list, only the first occurrence is removed.

Example:

* ```python
* # Create a list
* my\_list = [1, 2, 3, 2, 4, 5]
* # Remove the first occurrence of the value '2'
* my\_list.remove(2)
* print(my\_list) # Output: [1, 3, 2, 4, 5]
* ```
* It's important to note that if you want to remove an element from a list without knowing its index or value, you can use methods like `pop()` to remove the last element, or you can use list slicing to create a new list with the elements you want to keep. Additionally, you can use list comprehensions to filter out specific elements from a list and create a new list without them.

Python List Method

Python provides a variety of built-in list methods that allow you to manipulate and perform operations on lists efficiently. Here's a summary of some commonly used Python list methods along with examples:

1. \*\*append(element):\*\*

- The `append()` method is used to add an element to the end of a list.

\*\*Example:\*\*

```python

my\_list = [1, 2, 3]

my\_list.append(4)

print(my\_list) # Output: [1, 2, 3, 4]

```

2. \*\*extend(iterable):\*\*

- The `extend()` method is used to add elements from an iterable (e.g., another list) to the end of the list.

\*\*Example:\*\*

```python

my\_list = [1, 2, 3]

my\_list.extend([4, 5])

print(my\_list) # Output: [1, 2, 3, 4, 5]

```

3. \*\*insert(index, element):\*\*

- The `insert()` method allows you to insert an element at a specified index in the list.

\*\*Example:\*\*

```python

my\_list = [1, 2, 3]

my\_list.insert(1, 4)

print(my\_list) # Output: [1, 4, 2, 3]

```

4. \*\*remove(element):\*\*

- The `remove()` method deletes the first occurrence of a specific element in the list.

\*\*Example:\*\*

```python

my\_list = [1, 2, 3, 2, 4, 5]

my\_list.remove(2)

print(my\_list) # Output: [1, 3, 2, 4, 5]

```

5. pop([index]):

- The `pop()` method removes and returns the element at the specified index. If no index is provided, it removes and returns the last element.

Example:

```python

my\_list = [1, 2, 3, 4]

popped\_element = my\_list.pop(2)

print(popped\_element) # Output: 3

print(my\_list) # Output: [1, 2, 4]

```

6. \*\*index(element):\*\*

- The `index()` method returns the index of the first occurrence of a specified element.

\*\*Example:\*\*

```python

my\_list = [10, 20, 30, 20, 40]

index = my\_list.index(20)

print(index) # Output: 1

```

7. \*\*count(element):\*\*

- The `count()` method returns the number of times a specific element appears in the list.

\*\*Example:\*\*

```python

my\_list = [1, 2, 2, 3, 2, 4, 5]

count = my\_list.count(2)

print(count) # Output: 3

These are just a few of the many list methods available in Python for various list operations. Understanding and using these methods can greatly simplify list manipulation and make your code more efficient.

**Day 3**

**Errors and Exceptions.**

* In Python, errors and exceptions are part of the debugging landscape.
* Errors are issues that prevent the interpreter from running the program, while exceptions are raised during the program's execution.
* Common types of errors include syntax errors, which occur when the code doesn't adhere to the language's rules, and runtime errors, which happen during program execution.
* Exceptions, on the other hand, are events that occur during runtime but don't necessarily lead to a program crash.

**Syntax Errors.**

* These are the most common complains you get while you are still learning python.

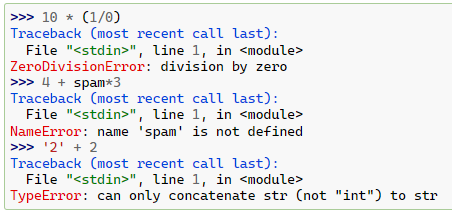
A screen shot of a computer code

Description automatically generated

**Exceptions**

* Errors detected during execution are called exceptions and are not unconditionally fatal.

Example



* The last line of the error message indicates what happened.
* Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError.
* The string printed as the exception type is the name of the built-in exception that occurred.
* This is true for all built-in exceptions but need not be true for user-defined exceptions (although it is a useful convention).
* Standard exception names are built-in identifiers (not reserved keywords).
* The rest of the line provides detail based on the type of exception and what caused it.
* The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback.
* In general, it contains a stack traceback listing source lines; however, it will not display lines read from standard input.
* Built-in Exceptions lists the built-in exceptions and their meanings.

**Handling Exceptions**

* To handle exceptions, Python provides a try-except block Code within the try block is executed, and if an exception occurs, the corresponding except block is triggered.
* This allows for graceful error handling and prevents your program from abruptly terminating.
* It’s possible to write programs that handles exceptions.

A white card with red text

Description automatically generated

Python also supports raising exceptions manually using the raise keyword. This can be useful for signaling specific conditions that should be handled by the calling code.

Understanding and effectively handling errors and exceptions is crucial for writing robust and reliable Python code.

**Syntax Error:**

python

**# Syntax Error**

**print("Hello, world!"**

```

Here, the missing closing parenthesis will raise a syntax error.

**Runtime Error:**

**python**

**# Runtime Error**

**result = 10 / 0**

This will result in a `ZeroDivisionError` during program execution.

**Handling Exceptions:**

python

# Handling Exceptions

try:

x = int(input("Enter a number: "))

result = 10 / x

print("Result:", result)

except ZeroDivisionError:

print("Cannot divide by zero!")

except ValueError:

print("Invalid input. Please enter a number.")

This code attempts to take user input, convert it to an integer, and then perform a division. If the user enters zero or a non-numeric value, it catches the corresponding exceptions.

**Raising Exceptions:**

python

# Raising Exceptions

def validate\_age(age):

if age < 0:

raise ValueError("Age cannot be negative.")

try:

user\_age = int(input("Enter your age: "))

validate\_age(user\_age)

print("Valid age!")

except ValueError as ve:

print(f"Error: {ve}")

In this example, the `validate\_age` function raises a `ValueError` if the age is negative. The main code then catches and handles this exception.

These examples illustrate how errors and exceptions work in Python, along with the use of try-except blocks and manual exception raising.

Raising an Exception

Raising an exception in Python involves using the `raise` keyword to signal that a specific error or exceptional condition has occurred during the program's execution. This can be useful for indicating problems or enforcing certain conditions in your code.

1. **Syntax:**

```python

raise ExceptionType("Error message")

```

**2. Custom Exceptions:**

You can create your own exception classes by inheriting from the `Exception` class or one of its subclasses.

```python

class CustomError(Exception):

pass

# Raise custom exception

raise CustomError("This is a custom error.")

```

3. **Raising Built-in Exceptions:**

You can raise built-in exceptions with specific error messages to convey information about the problem.

python

raise ValueError("Invalid input. Please enter a positive number.")

4. **Conditional Raising:**

You can use conditional statements to raise exceptions based on certain conditions.

```python

x = -5

if x < 0:

raise ValueError("Value must be non-negative.")

```

**5. Handling Raised Exceptions:**

It's important to use try-except blocks to catch and handle exceptions appropriately. This prevents the program from crashing and allows for controlled error management.

```python

try:

# code that may raise an exception

except CustomError as ce:

print(f"Caught a custom exception: {ce}")

except Exception as e:

print(f"Caught a general exception: {e}")

**Raising exceptions**

* is a powerful mechanism for signaling exceptional conditions in your code and providing meaningful error messages for debugging and user feedback.
* Raising an exception in Python involves using the `raise` keyword to deliberately generate an error or signal a specific condition. Here's a concise summary:

1. Basic Syntax:

- Use the `raise` keyword followed by the type of exception you want to raise.

- Optionally, provide an error message to provide more context.

```python

raise ValueError("This is a custom error message.")

```

2. \*\*Custom Exceptions:\*\*

- You can create your own exception classes by inheriting from built-in exception classes or the base `Exception` class.

```python

class CustomError(Exception):

pass

raise CustomError("Custom error message.")

```

3. \*\*Conditional Raising:\*\*

- You can use conditional statements to decide when to raise an exception.

```python

x = -5

if x < 0:

raise ValueError("Value must be non-negative.")

```

4. \*\*Handling Raised Exceptions:\*\*

- Use try-except blocks to catch and handle exceptions.

- This prevents the program from crashing and allows for graceful error handling.

```python

try:

# code that may raise an exception

except CustomError as ce:

print(f"Caught a custom exception: {ce}")

except Exception as e:

print(f"Caught a general exception: {e}")

```

Raising exceptions is a controlled way to handle errors in your code, providing a mechanism for communicating issues and enabling proper error management.

**Explaining exceptions**

in Python involves understanding how the language deals with errors during program execution.

Here's a brief summary:

1. \*\*Types of Errors:\*\*

- \*\*Syntax Errors:\*\* Detected by the interpreter during the parsing of the code. They prevent the program from running.

- \*\*Runtime Errors (Exceptions):\*\* Occur during the execution of the program, leading to its interruption.

2. \*\*Exception Handling:\*\*

- Python uses a try-except block for handling exceptions.

- Code within the try block is executed, and if an exception occurs, the corresponding except block is triggered.

3. \*\*Common Exceptions:\*\*

- Examples include `TypeError`, `ValueError`, `ZeroDivisionError`, and more.

- Each exception type corresponds to a specific error scenario.

4. \*\*Custom Exceptions:\*\*

- Developers can create custom exception classes by inheriting from built-in exception classes or the base `Exception` class.

```python

class CustomError(Exception):

pass

```

5. \*\*Raising Exceptions:\*\*

- Developers can manually raise exceptions using the `raise` keyword.

- This is useful for signaling specific conditions or errors within the code.

```python

raise ValueError("Invalid input.")

```

6. \*\*Exception Hierarchy:\*\*

- Exceptions are organized in a hierarchy, with the base class being `BaseException`.

- Specific exception classes inherit from more general ones.

7. \*\*Finally Block:\*\*

- The `finally` block, if included, is executed regardless of whether an exception is raised or not. It's often used for cleanup operations.

```python

try:

# code that may raise an exception

except SomeException:

# handle specific exception

finally:

# code that will always run

```

Understanding exceptions is crucial for writing robust code, as it allows for controlled error handling and ensures that unexpected issues don't lead to program crashes.

**Day4**

**Classes**

In Python, classes serve as blueprints for creating objects, instances of the class. They enable the bundling of data (attributes) and functionality (methods). Here's a concise overview:

- \*\*Defining a Class:\*\*

- Use the `class` keyword followed by the class name and a colon.

- Class names are conventionally written in CamelCase.

```python

class MyClass:

def \_\_init\_\_(self, var1, var2):

self.var1 = var1

self.var2 = var2

```

- \*\*Class Methods:\*\*

- Methods are functions defined within a class, operating on the class's data.

```python

class MyClass:

def \_\_init\_\_(self, var1, var2):

self.var1 = var1

self.var2 = var2

def display\_vars(self):

print("Var1:", self.var1)

print("Var2:", self.var2)

```

- \*\*Creating an Object:\*\*

- Instantiate a class to create an object.

- Call methods on the object.

```python

obj = MyClass(10, 20)

obj.display\_vars()

```

- \*\*Inheritance:\*\*

- Allows a class (subclass) to inherit properties and methods from another class (superclass).

```python

class ParentClass:

def \_\_init\_\_(self, var1, var2):

self.var1 = var1

self.var2 = var2

class ChildClass(ParentClass):

def \_\_init\_\_(self, var1, var2, var3):

super().\_\_init\_\_(var1, var2)

self.var3 = var3

```

- \*\*Principles of OOP:\*\*

- \*\*Encapsulation:\*\* Bundling data and methods into a class, controlling access through public methods. Use underscores for private and protected attributes.

```python

class MyClass:

def \_\_init\_\_(self):

self.\_protected\_var = 10 # protected variable

self.\_\_private\_var = 20 # private variable

def get\_private\_var(self):

return self.\_\_private\_var

```

- \*\*Abstraction:\*\* Hiding complex implementation details, often achieved through abstract classes and interfaces.

```python

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass

class Square(Shape):

def \_\_init\_\_(self, side):

self.side = side

def area(self):

return self.side \* self.side

```

- \*\*Polymorphism:\*\* Objects of different classes treated as objects of a common superclass, achieved through method overriding.

```python

class Animal:

def sound(self):

pass

class Dog(Animal):

def sound(self):

return "Woof!"

class Cat(Animal):

def sound(self):

return "Meow!"

def print\_animal\_sound(animal):

print(animal.sound())

dog = Dog()

cat = Cat()

print\_animal\_sound(dog)

print\_animal\_sound(cat)

```

These principles promote organized, efficient, and maintainable Python code. Additionally, Python's approach to namespaces and scopes enhances code clarity, and the proper use of class and instance variables contributes to effective object-oriented programming.