**WEEK 3**

**DATA STRUCTERES**

**DAY 1 DATA STRUCTURES**

CREATING LIST

How to create a list?

In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

# empty list

my\_list = []

# list of integers

my\_list = [1, 2, 3]

# list with mixed data types

my\_list = [1, "Hello", 3.4]

A list can also have another list as an item. This is called a nested list.

# nested list

my\_list = ["mouse", [8, 4, 6], ['a']]

MORE ON LIST

Access List Elements

There are various ways in which we can access the elements of a list.

List Index

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.

# List indexing

my\_list = ['p', 'r', 'o', 'b', 'e']

# Output: p

print(my\_list[0])

# Output: o

print(my\_list[2])

# Output: e

print(my\_list[4])

# Nested List

n\_list = ["Happy", [2, 0, 1, 5]]

# Nested indexing

print(n\_list[0][1])

print(n\_list[1][3])

# Error! Only integer can be used for indexing

print(my\_list[4.0])

Output

p

o

e

a

5

Traceback (most recent call last):

File "<string>", line 21, in <module>

TypeError: list indices must be integers or slices, not float.

Using Lists as Stacks and Queues

Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

# Negative indexing in lists

my\_list = ['p','r','o','b','e']

print(my\_list[-1])

print(my\_list[-5])

When we run the above program, we will get the following output:

e

p

Python list indexingList indexing in Python.

SLICING LIST IN PYTHON

How to slice lists in Python?

We can access a range of items in a list by using the slicing operator :(colon).

# List slicing in Python

my\_list = ['p','r','o','g','r','a','m','i','z']

# elements 3rd to 5th

print(my\_list[2:5])

# elements beginning to 4th

print(my\_list[:-5])

# elements 6th to end

print(my\_list[5:])

# elements beginning to end

print(my\_list[:])

Output

['o', 'g', 'r']

['p', 'r', 'o', 'g']

['a', 'm', 'i', 'z']

['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two indices that will slice that portion from the list.

ADD/CHANGE LIST ELEMENTS

Add/Change List Elements

Lists are mutable, meaning their elements can be changed unlike string or tuple.

We can use the assignment operator = to change an item or a range of items.

# Correcting mistake values in a list

odd = [2, 4, 6, 8]

# change the 1st item

odd[0] = 1

print(odd)

# change 2nd to 4th items

odd[1:4] = [3, 5, 7]

print(odd)

Output

[1, 4, 6, 8]

[1, 3, 5, 7]

We can add one item to a list using the append() method or add several items using extend() method.

# Appending and Extending lists in Python

odd = [1, 3, 5]

odd.append(7)

print(odd)

odd.extend([9, 11, 13])

print(odd)

Output

[1, 3, 5, 7]

[1, 3, 5, 7, 9, 11, 13]

We can also use + operator to combine two lists. This is also called concatenation.

The \* operator repeats a list for the given number of times.

# Concatenating and repeating lists

odd = [1, 3, 5]

print(odd + [9, 7, 5])

print(["re"] \* 3)

Output

[1, 3, 5, 9, 7, 5]

['re', 're', 're']

Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

# Demonstration of list insert() method

odd = [1, 9]

odd.insert(1,3)

print(odd)

odd[2:2] = [5, 7]

print(odd)

Output

[1, 3, 9]

[1, 3, 5, 7, 9]

The del statement / Tuples and Sequences

Delete/Remove List Elements

We can delete one or more items from a list using the keyword del. It can even delete the list entirely.

# Deleting list items

my\_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# delete one item

del my\_list[2]

print(my\_list)

# delete multiple items

del my\_list[1:5]

print(my\_list)

# delete entire list

del my\_list

# Error: List not defined

print(my\_list)

Output

['p', 'r', 'b', 'l', 'e', 'm']

['p', 'm']

Traceback (most recent call last):

File "<string>", line 18, in <module>

NameError: name 'my\_list' is not defined

We can use remove() method to remove the given item or pop() method to remove an item at the given index.

The pop() method removes and returns the last item if the index is not provided. This helps us implement lists as stacks (first in, last out data structure).

We can also use the clear() method to empty a list.

my\_list = ['p','r','o','b','l','e','m']

my\_list.remove('p')

# Output: ['r', 'o', 'b', 'l', 'e', 'm']

print(my\_list)

# Output: 'o'

print(my\_list.pop(1))

# Output: ['r', 'b', 'l', 'e', 'm']

print(my\_list)

# Output: 'm'

print(my\_list.pop())

# Output: ['r', 'b', 'l', 'e']

print(my\_list)

my\_list.clear()

# Output: []

print(my\_list)

Output

['r', 'o', 'b', 'l', 'e', 'm']

o

['r', 'b', 'l', 'e', 'm']

m

['r', 'b', 'l', 'e']

[]

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

>>> my\_list = ['p','r','o','b','l','e','m']

>>> my\_list[2:3] = []

>>> my\_list

['p', 'r', 'b', 'l', 'e', 'm']

>>> my\_list[2:5] = []

>>> my\_list

['p', 'r', 'm']

Python List Methods

Methods that are available with list objects in Python programming are tabulated below.

They are accessed as list.method(). Some of the methods have already been used above.

Python List Methods

append() - Add an element to the end of the list

extend() - Add all elements of a list to the another list

insert() - Insert an item at the defined index

remove() - Removes an item from the list

pop() - Removes and returns an element at the given index

clear() - Removes all items from the list

index() - Returns the index of the first matched item

count() - Returns the count of the number of items passed as an argument

sort() - Sort items in a list in ascending order

reverse() - Reverse the order of items in the list.

copy() - Returns a shallow copy of the list

Some examples of Python list methods:

# Python list methods

my\_list = [3, 8, 1, 6, 0, 8, 4]

# Output: 1

print(my\_list.index(8))

# Output: 2

print(my\_list.count(8))

my\_list.sort()

# Output: [0, 1, 3, 4, 6, 8, 8]

print(my\_list)

my\_list.reverse()

# Output: [8, 8, 6, 4, 3, 1, 0]

print(my\_list)

Output

1

2

[0, 1, 3, 4, 6, 8, 8]

[8, 8, 6, 4, 3, 1, 0]

List Comprehension: Elegant way to create Lists

List comprehension is an elegant and concise way to create a new list from an existing list in Python.

A list comprehension consists of an expression followed by for statement inside square brackets.

Here is an example to make a list with each item being increasing power of 2.

pow2 = [2 \*\* x for x in range(10)]

print(pow2)

Output

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

This code is equivalent to:

pow2 = []

for x in range(10):

pow2.append(2 \*\* x)

A list comprehension can optionally contain more for or if statements. An optional if statement can filter out items for the new list. Here are some examples.

>>> pow2 = [2 \*\* x for x in range(10) if x > 5]

>>> pow2

[64, 128, 256, 512]

>>> odd = [x for x in range(20) if x % 2 == 1]

>>> odd

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

>>> [x+y for x in ['Python ','C '] for y in ['Language','Programming']]

['Python Language', 'Python Programming', 'C Language', 'C Programming']

Other List Operations in Python

List Membership Test

We can test if an item exists in a list or not, using the keyword

my\_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# Output: True

print('p' in my\_list)

# Output: False

print('a' in my\_list)

# Output: True

print('c' not in my\_list)

Output

True

False

True

Iterating Through a List

Using a for loop we can iterate through each item in a list.

for fruit in ['apple','banana','mango']:

print("I like",fruit)

Output

I like apple

I like banana

I like mango

Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

>>>

>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}

>>> for k, v in knights.items():

... print(k, v)

...

gallahad the pure

robin the brave

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

>>>

>>> for i, v in enumerate(['tic', 'tac', 'toe']):

... print(i, v)

...

0 tic

1 tac

2 toe

To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

>>>

>>> questions = ['name', 'quest', 'favorite color']

>>> answers = ['lancelot', 'the holy grail', 'blue']

>>> for q, a in zip(questions, answers):

... print('What is your {0}? It is {1}.'.format(q, a))

...

What is your name? It is lancelot.

What is your quest? It is the holy grail.

What is your favorite color? It is blue.

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

>>>

>>> for i in reversed(range(1, 10, 2)):

... print(i)

...

9

7

5

3

1

To loop over a sequence in sorted order, use the sorted() function which returns a new sorted list while leaving the source unaltered.

>>>

>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

>>> for f in sorted(set(basket)):

... print(f)

...

apple

banana

orange

Pear

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

>>>

>>> import math

>>> raw\_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]

>>> filtered\_data = []

>>> for value in raw\_data:

... if not math.isnan(value):

... filtered\_data.append(value)

...

>>> filtered\_data

[56.2, 51.7, 55.3, 52.5, 47.8]

More on conditions

The conditions used in while and if statements can contain any operators, not just comparisons.

The comparison operators in and not in check whether a value occurs (does not occur) in a sequence. The operators is and is not compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, a < b == c tests whether a is less than b and moreover b equals c.

Comparisons may be combined using the Boolean operators and and or, and the outcome of a comparison (or of any other Boolean expression) may be negated with not. These have lower priorities than comparison operators; between them, not has the highest priority and or the lowest, so that A and not B or C is equivalent to (A and (not B)) or C. As always, parentheses can be used to express the desired composition.

The Boolean operators and and or are so-called short-circuit operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if A and C are true but B is false, A and B and C does not evaluate the expression C. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

>>>

>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'

>>> non\_null = string1 or string2 or string3

>>> non\_null

'Trondheim'

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing = in an expression when == was intended.

Comparing Sequences and Other Types

In Python, you can compare sequences (such as strings, lists, and tuples) and other types (such as integers and floats) using various comparison operators. These operators allow you to check the relationship between values and make decisions in your code. Here are the common comparison operators used in Python:

1. \*\*Equality (`==`)\*\*: Checks if two values are equal.

```python

x == y # True if x is equal to y

```

2. \*\*Inequality (`!=`)\*\*: Checks if two values are not equal.

```python

x != y # True if x is not equal to y

```

3. \*\*Less than (`<`)\*\*: Checks if the left operand is less than the right operand.

```python

x < y # True if x is less than y

```

4. \*\*Less than or equal to (`<=`)\*\*: Checks if the left operand is less than or equal to the right operand.

```python

x <= y # True if x is less than or equal to y

```

5. \*\*Greater than (`>`)\*\*: Checks if the left operand is greater than the right operand.

```python

x > y # True if x is greater than y

```

6. \*\*Greater than or equal to (`>=`)\*\*: Checks if the left operand is greater than or equal to the right operand.

```python

x >= y # True if x is greater than or equal to y

```

When comparing sequences (e.g., strings, lists, tuples), these operators consider the lexicographical (dictionary) order. The elements of the sequence are compared element by element until a difference is found or the entire sequence is compared.

For example, when comparing strings:

```python

"apple" < "banana" # True, because "apple" comes before "banana" in the dictionary

"apple" == "banana" # False, because the strings are not equal

```

When comparing other types like integers and floats, these operators work as expected:

```python

5 < 10 # True, because 5 is less than 10

3.14 >= 2.71 # True, because 3.14 is greater than or equal to 2.71

```

You can use these comparison operators to make conditional statements, such as `if` and `while` statements, to control the flow of your program based on the results of these comparisons.

**DAY 2 PYTHON LIST**

CREATING LIST

How to create a list?

In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

# empty list

my\_list = []

# list of integers

my\_list = [1, 2, 3]

# list with mixed data types

my\_list = [1, "Hello", 3.4]

A list can also have another list as an item. This is called a nested list.

# nested list

my\_list = ["mouse", [8, 4, 6], ['a']]

MORE ON LIST

Access List Elements

There are various ways in which we can access the elements of a list.

List Index

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.

# List indexing

my\_list = ['p', 'r', 'o', 'b', 'e']

# Output: p

print(my\_list[0])

# Output: o

print(my\_list[2])

# Output: e

print(my\_list[4])

# Nested List

n\_list = ["Happy", [2, 0, 1, 5]]

# Nested indexing

print(n\_list[0][1])

print(n\_list[1][3])

# Error! Only integer can be used for indexing

print(my\_list[4.0])

Output

p

o

e

a

5

Traceback (most recent call last):

File "<string>", line 21, in <module>

TypeError: list indices must be integers or slices, not float.

Using Lists as Stacks and Queues

Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

# Negative indexing in lists

my\_list = ['p','r','o','b','e']

print(my\_list[-1])

print(my\_list[-5])

When we run the above program, we will get the following output:

e

p

Python list indexingList indexing in Python.

ADD/CHANGE LIST ELEMENTS

Add/Change List Elements

Lists are mutable, meaning their elements can be changed unlike string or tuple.

We can use the assignment operator = to change an item or a range of items.

# Correcting mistake values in a list

odd = [2, 4, 6, 8]

# change the 1st item

odd[0] = 1

print(odd)

# change 2nd to 4th items

odd[1:4] = [3, 5, 7]

print(odd)

Output

[1, 4, 6, 8]

[1, 3, 5, 7]

We can add one item to a list using the append() method or add several items using extend() method.

# Appending and Extending lists in Python

odd = [1, 3, 5]

odd.append(7)

print(odd)

odd.extend([9, 11, 13])

print(odd)

Output

[1, 3, 5, 7]

[1, 3, 5, 7, 9, 11, 13]

We can also use + operator to combine two lists. This is also called concatenation.

The \* operator repeats a list for the given number of times.

# Concatenating and repeating lists

odd = [1, 3, 5]

print(odd + [9, 7, 5])

print(["re"] \* 3)

Output

[1, 3, 5, 9, 7, 5]

['re', 're', 're']

Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

# Demonstration of list insert() method

odd = [1, 9]

odd.insert(1,3)

print(odd)

odd[2:2] = [5, 7]

print(odd)

Output

[1, 3, 9]

[1, 3, 5, 7, 9]

Delete/Remove List Elements

We can delete one or more items from a list using the keyword del. It can even delete the list entirely.

# Deleting list items

my\_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# delete one item

del my\_list[2]

print(my\_list)

# delete multiple items

del my\_list[1:5]

print(my\_list)

# delete entire list

del my\_list

# Error: List not defined

print(my\_list)

Output

['p', 'r', 'b', 'l', 'e', 'm']

['p', 'm']

Traceback (most recent call last):

File "<string>", line 18, in <module>

NameError: name 'my\_list' is not defined

We can use remove() method to remove the given item or pop() method to remove an item at the given index.

The pop() method removes and returns the last item if the index is not provided. This helps us implement lists as stacks (first in, last out data structure).

We can also use the clear() method to empty a list.

my\_list = ['p','r','o','b','l','e','m']

my\_list.remove('p')

# Output: ['r', 'o', 'b', 'l', 'e', 'm']

print(my\_list)

# Output: 'o'

print(my\_list.pop(1))

# Output: ['r', 'b', 'l', 'e', 'm']

print(my\_list)

# Output: 'm'

print(my\_list.pop())

# Output: ['r', 'b', 'l', 'e']

print(my\_list)

my\_list.clear()

# Output: []

print(my\_list)

Output

['r', 'o', 'b', 'l', 'e', 'm']

o

['r', 'b', 'l', 'e', 'm']

m

['r', 'b', 'l', 'e']

[]

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

>>> my\_list = ['p','r','o','b','l','e','m']

>>> my\_list[2:3] = []

>>> my\_list

['p', 'r', 'b', 'l', 'e', 'm']

>>> my\_list[2:5] = []

>>> my\_list

['p', 'r', 'm']

Python List Methods

Methods that are available with list objects in Python programming are tabulated below.

They are accessed as list.method(). Some of the methods have already been used above.

Python List Methods

append() - Add an element to the end of the list

extend() - Add all elements of a list to the another list

insert() - Insert an item at the defined index

remove() - Removes an item from the list

pop() - Removes and returns an element at the given index

clear() - Removes all items from the list

index() - Returns the index of the first matched item

count() - Returns the count of the number of items passed as an argument

sort() - Sort items in a list in ascending order

reverse() - Reverse the order of items in the list.

copy() - Returns a shallow copy of the list

Some examples of Python list methods:

# Python list methods

my\_list = [3, 8, 1, 6, 0, 8, 4]

# Output: 1

print(my\_list.index(8))

# Output: 2

print(my\_list.count(8))

my\_list.sort()

# Output: [0, 1, 3, 4, 6, 8, 8]

print(my\_list)

my\_list.reverse()

# Output: [8, 8, 6, 4, 3, 1, 0]

print(my\_list)

Output

1

2

[0, 1, 3, 4, 6, 8, 8]

[8, 8, 6, 4, 3, 1, 0]

List Comprehension: Elegant way to create Lists

List comprehension is an elegant and concise way to create a new list from an existing list in Python.

A list comprehension consists of an expression followed by for statement inside square brackets.

Here is an example to make a list with each item being increasing power of 2.

pow2 = [2 \*\* x for x in range(10)]

print(pow2)

Output

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

This code is equivalent to:

pow2 = []

for x in range(10):

pow2.append(2 \*\* x)

A list comprehension can optionally contain more for or if statements. An optional if statement can filter out items for the new list. Here are some examples.

>>> pow2 = [2 \*\* x for x in range(10) if x > 5]

>>> pow2

[64, 128, 256, 512]

>>> odd = [x for x in range(20) if x % 2 == 1]

>>> odd

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

>>> [x+y for x in ['Python ','C '] for y in ['Language','Programming']]

['Python Language', 'Python Programming', 'C Language', 'C Programming']

Other List Operations in Python

List Membership Test

We can test if an item exists in a list or not, using the keyword in.

my\_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# Output: True

print('p' in my\_list)

# Output: False

print('a' in my\_list)

# Output: True

print('c' not in my\_list)

Output

True

False

True

Iterating Through a List

Using a for loop we can iterate through each item in a list.

for fruit in ['apple','banana','mango']:

print("I like",fruit)

Output

I like apple

I like banana

I like mango

**DAY 3 ERRORS AND EXCEPTION**

ERRORS AND EXCEPTION

Errors and exceptions are a common part of programming in Python. They occur when the interpreter encounters a problem while running your code. Understanding different types of errors and exceptions is essential for debugging and writing reliable code. Here are some common categories of errors and exceptions in Python:

1. \*\*Syntax Errors\*\*:

- Syntax errors occur when the code is not written in a valid Python syntax.

- These errors are detected during the parsing phase, before the code is executed.

- Example: Missing colons, unmatched parentheses, invalid indentation.

2. \*\*Runtime Errors (Exceptions)\*\*:

- Runtime errors, also known as exceptions, occur during the execution of the code.

- These errors can be handled using try-except blocks.

- Common exceptions include `ZeroDivisionError`, `IndexError`, `TypeError`, `ValueError`, and more.

3. \*\*Logical Errors\*\*:

- Logical errors are not detected by the Python interpreter, but they lead to incorrect program behavior.

- These errors are typically caused by incorrect program logic or algorithm design.

Here are some examples of common exceptions in Python:

- \*\*`ZeroDivisionError`\*\*: Raised when division or modulo operation is performed with zero as the divisor.

- \*\*`IndexError`\*\*: Raised when an index is out of range in a sequence (e.g., list or tuple).

- \*\*`TypeError`\*\*: Raised when an operation is performed on an object of inappropriate data type.

- \*\*`ValueError`\*\*: Raised when a built-in function receives an argument of the correct data type but with an inappropriate value.

- \*\*`NameError`\*\*: Raised when an identifier is not found in the local or global namespace.

- \*\*`FileNotFoundError`\*\*: Raised when trying to open a file that does not exist.

To handle exceptions in Python, you can use the `try` and `except` blocks:

```python

try:

# Code that may raise an exception

result = 10 / 0

except ZeroDivisionError:

# Code to handle the exception

print("Error: Division by zero")

```

You can also use `else` and `finally` blocks with `try` and `except` for more complex exception handling:

```python

try:

# Code that may raise an exception

result = 10 / 0

except ZeroDivisionError:

# Code to handle the exception

print("Error: Division by zero")

else:

# Code to run if no exception occurred

print("Result:", result)

finally:

# Code that runs whether an exception occurred or not

print("Finally block")

```

Handling exceptions allows your program to recover gracefully from errors and continue executing code, rather than crashing. It is important to use exceptions appropriately to make your code more robust and reliable.

RAISING AN EXCEPTION

In Python, you can raise your own exceptions to signal an error or exceptional condition in your code. This can be useful for handling custom error scenarios or situations where built-in exceptions may not be appropriate. To raise an exception, you can use the `raise` statement.

Here's how you can raise an exception in Python:

```python

raise Exception("This is a custom exception message.")

```

In this example, we're raising a generic `Exception` and providing a custom error message as a string. You can replace `"This is a custom exception message."` with any error message that conveys the reason for the exception.

If you want to create a custom exception class, you can define a new class that inherits from the built-in `Exception` class or one of its subclasses. Here's an example of creating a custom exception:

```python

class MyCustomException(Exception):

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

super().\_\_init\_\_(message)

# Raise the custom exception

raise MyCustomException("This is a custom exception message.")

```

In this example, we've defined a `MyCustomException` class that inherits from `Exception` and takes a custom error message as an argument.

You can catch and handle custom exceptions, along with built-in exceptions, using `try` and `except` blocks. Here's an example of catching and handling a custom exception:

```python

try:

raise MyCustomException("Something went wrong!")

except MyCustomException as e:

print(f"Custom exception caught: {e}")

except Exception as e:

print(f"Other exception caught: {e}")

```

In this code, the custom exception is caught in the first `except` block, and its error message is printed. If an exception of a different type occurs, it is caught by the second `except` block.

Raising custom exceptions is helpful for improving the clarity of your code and making it easier to handle specific error scenarios in a structured way.

**DAY 4 CLASSES**

**CLASSES**

In Python, classes are a fundamental part of object-oriented programming (OOP). They allow you to define a blueprint for creating objects, which are instances of the class. Classes are used to encapsulate data and behavior into a single unit, making your code more organized and easier to maintain.

Here's a basic overview of how to define and use classes in Python:

1. Class Definition:

You can define a class using the `class` keyword, followed by the class name. The class definition typically includes attributes (variables) and methods (functions) that operate on those attributes.

```python

class MyClass:

# Class attributes (shared by all instances)

class\_attribute = 0

# Constructor method (initialize instance-specific attributes)

def \_\_init\_\_(self, attribute1, attribute2):

self.attribute1 = attribute1

self.attribute2 = attribute2

# Instance method

def instance\_method(self):

return f"Instance method called with attribute1 = {self.attribute1}"

# Class method (operates on class-level data)

@classmethod

def class\_method(cls):

return f"Class method called with class\_attribute = {cls.class\_attribute}"

# Static method (independent of instance or class data)

@staticmethod

def static\_method():

return "Static method called"

```

2. Creating Instances:

To create an instance of a class, you can simply call the class as if it were a function, passing any required arguments to the constructor (the `\_\_init\_\_` method):

```python

obj1 = MyClass(1, 2)

obj2 = MyClass(3, 4)

```

3. Accessing Attributes and Methods:

You can access the attributes and methods of an instance using the dot notation:

```python

print(obj1.attribute1) # Accessing an instance attribute

print(obj1.instance\_method()) # Calling an instance method

print(obj2.class\_attribute) # Accessing a class attribute

print(MyClass.class\_method()) # Calling a class method

print(MyClass.static\_method()) # Calling a static method

```

4. Inheritance:

Python supports class inheritance, which allows you to create a new class based on an existing class. The new class can inherit the attributes and methods of the parent class and can also override or extend them.

```python

class ChildClass(MyClass):

def child\_method(self):

return "Child method called"

child\_obj = ChildClass(5, 6)

print(child\_obj.child\_method())

```

5. Encapsulation, Polymorphism, and Abstraction:

OOP principles like encapsulation (hiding data), polymorphism (ability to use objects of different classes interchangeably), and abstraction (simplifying complex systems) are supported and encouraged in Python through the use of classes and objects.

This is just a basic introduction to Python classes. You can create more complex class hierarchies and structures as your application's needs dictate. Classes are a powerful tool for structuring and organizing your code in a modular and maintainable way.

PYTHON SCOPES AND NAMESPACES

In Python, scopes and namespaces are essential concepts that determine how variables are accessed and organized in your code. Understanding these concepts is crucial for writing clean and error-free Python code. Let's dive into Python's scopes and namespaces:

1. \*\*Namespace\*\*:

A namespace is a container that holds a mapping of names (identifiers) to objects. Namespaces are used to organize and manage variables, functions, classes, and other objects in Python.

Python has several types of namespaces, including:

- \*\*Local Namespace\*\*: This namespace contains variables defined within a function. It is temporary and only exists during the execution of the function.

- \*\*Enclosing Namespace\*\*: In nested functions, each inner function has access to the names defined in its outer or enclosing function.

- \*\*Global Namespace\*\*: The global namespace contains names defined at the top level of a module or script. Variables defined in the global namespace are accessible from anywhere within the module.

- \*\*Built-in Namespace\*\*: This namespace contains the names of built-in functions, types, and exceptions, like `print()`, `len()`, and `int`.

2. \*\*Scope\*\*:

A scope is a region in your code where a particular namespace is directly accessible. Scopes can be nested, meaning that an inner scope has access to names from the outer scope, but not vice versa. The order in which Python searches for names in different scopes is known as the "LEGB" rule:

- \*\*Local\*\*: Names defined in the current function's local namespace.

- \*\*Enclosing\*\*: Names defined in the local namespace of any enclosing functions.

- \*\*Global\*\*: Names defined at the module level (global namespace).

- \*\*Built-in\*\*: Names in the built-in namespace.

Here's a practical example:

```python

x = 10 # This is in the global namespace

def my\_function():

y = 5 # This is in the local namespace of my\_function

print(x) # Access the global variable x

def inner\_function():

z = 2 # This is in the local namespace of inner\_function

print(x) # Access the global variable x

print(y) # Access the variable y from the enclosing function

print(z)

inner\_function()

my\_function()

```

In this example, `x` is in the global scope and is accessible from both `my\_function` and `inner\_function`. `y` is in the local scope of `my\_function` and is accessible from both `my\_function` and `inner\_function` due to the enclosing scope. `z` is only accessible within `inner\_function` because it is defined in its local scope.

If you ever need to modify a variable from an outer (e.g., global) scope within a function, you can use the `global` keyword to indicate that you want to work with the global variable.

Understanding Python's scopes and namespaces is crucial for writing maintainable and bug-free code, as it ensures that variables are used in the intended context and don't lead to unexpected behavior.

MORE ON CLASSES

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

The simplest form of class definition looks like this:

class ClassName:

<statement-1>

.

.

.

<statement-N>

Class definitions, like function definitions (def statements), must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we’ll come back to this later. The function definitions inside a class normally have a peculiar form of an argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the namespace created by the class definition; we’ll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (ClassName in the example).

Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class’s namespace when the class object was created. So, if the class definition looked like this:

class MyClass:

"""A simple example class"""

i = 12345

def f(self):

return 'hello world'

then MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of MyClass.i by assignment. \_\_doc\_\_ is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

x = MyClass()

creates a new instance of the class and assigns this object to the local variable x.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named \_\_init\_\_(), like this:

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

self.data = []

When a class defines an \_\_init\_\_() method, class instantiation automatically invokes \_\_init\_\_() for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

x = MyClass()

Of course, the \_\_init\_\_() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to \_\_init\_\_(). For example,

>>>

>>> class Complex:

... def \_\_init\_\_(self, realpart, imagpart):

... self.r = realpart

... self.i = imagpart

...

>>> x = Complex(3.0, -4.5)

>>> x.r, x.i

(3.0, -4.5)

Instance Objects

Now, what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes, and methods.

data attributes correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if x is the instance of MyClass created above, the following piece of code will print the value 16, without leaving a trace:

x.counter = 1

while x.counter < 10:

x.counter = x.counter \* 2

print(x.counter)

del x.counter

The other kind of instance attribute reference is a method. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, x.f is a valid method reference, since MyClass.f is a function, but x.i is not, since MyClass.i is not. But x.f is not the same thing as MyClass.f — it is a method object, not a function object.

Method Objects

Usually, a method is called right after it is bound:

x.f()

In the MyClass example, this will return the string 'hello world'. However, it is not necessary to call a method right away: x.f is a method object, and can be stored away and called at a later time. For example:

xf = x.f

while True:

print(xf())

will continue to print hello world until the end of time.

What exactly happens when a method is called? You may have noticed that x.f() was called without an argument above, even though the function definition for f() specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used…

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call x.f() is exactly equivalent to MyClass.f(x). In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s instance object before the first argument.

If you still don’t understand how methods work, a look at the implementation can perhaps clarify matters. When a non-data attribute of an instance is referenced, the instance’s class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

CLASS AND INSTANCE VARIABLES

In Python, class variables and instance variables are used to store data within classes. They serve different purposes and have different scopes and lifetimes. Let's explore the differences between these two types of variables:

1. \*\*Class Variables\*\*:

- \*\*Scope\*\*: Class variables are shared among all instances (objects) of a class. They belong to the class itself, not to any specific instance.

- \*\*Declaration\*\*: You define class variables inside the class but outside any method. They are usually placed at the top of the class definition.

- \*\*Access\*\*: Class variables are accessed using the class name or an instance of the class.

Example:

```python

class MyClass:

class\_variable = 0 # This is a class variable

# Accessing a class variable

print(MyClass.class\_variable) # Using the class name

obj1 = MyClass()

obj2 = MyClass()

print(obj1.class\_variable) # Using an instance (also valid)

```

All instances of the class share the same value of the class variable.

2. \*\*Instance Variables\*\*:

- \*\*Scope\*\*: Instance variables are specific to each instance of a class. They are defined inside the class's methods (e.g., the constructor) using the `self` keyword.

- \*\*Declaration\*\*: You declare instance variables within instance methods, typically within the `\_\_init\_\_` constructor, which initializes them for each instance.

- \*\*Access\*\*: Instance variables are accessed through the instance (object) of the class, using the `self` keyword.

Example:

```python

class Person:

def \_\_init\_\_(self, name, age):

self.name = name # This is an instance variable

self.age = age # This is also an instance variable

person1 = Person("Alice", 30)

person2 = Person("Bob", 25)

print(person1.name) # Accessing instance variable

print(person2.age)

```

Each instance of the class has its own set of instance variables, which can have different values.

It's important to differentiate between class variables and instance variables based on whether the data should be shared among all instances of the class or specific to each instance. Class variables are useful for storing data that is common to all objects of the class, while instance variables are used for storing data that varies from instance to instance.

Here are some key takeaways:

- Class variables are shared across all instances and are defined outside any method in the class.

- Instance variables are specific to each instance and are defined within methods using the `self` keyword.

- Access class variables using the class name or an instance, and access instance variables using the instance itself.

Understanding when to use class variables and instance variables is essential for designing object-oriented Python programs effectively.

RANDOM REMARKS

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users (“clients”) of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called self. This is nothing more than a convention: the name self has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a class browser program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

# Function defined outside the class

def f1(self, x, y):

return min(x, x+y)

class C:

f = f1

def g(self):

return 'hello world'

h = g

Now f, g, and h are all attributes of class C that refer to function objects, and consequently, they are all methods of instances of C — h being exactly equivalent to g. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the self argument:

class Bag:

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

self.data = []

def add(self, x):

self.data.append(x)

def addtwice(self, x):

self.add(x)

self.add(x)

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we’ll find some good reasons why a method would want to reference its own class.

Each value is an object and therefore has a class (also called its type). It is stored as object.\_\_class\_\_.

PYTHON CLASSES

Python classes are fundamental to object-oriented programming (OOP). They allow you to create blueprints for objects, which are instances of those classes. Objects can have attributes (data) and methods (functions) associated with them. Here's an overview of Python classes and how to use them:

1. \*\*Defining a Class\*\*:

You can define a class in Python using the `class` keyword followed by the class name. Typically, the class definition includes attributes and methods.

```python

class MyClass:

# Class attributes (shared among all instances)

class\_variable = 0

# Constructor method (initialize instance-specific attributes)

def \_\_init\_\_(self, attribute1, attribute2):

self.attribute1 = attribute1

self.attribute2 = attribute2

# Instance method

def instance\_method(self):

return f"Instance method called with attribute1 = {self.attribute1}"

```

2. \*\*Creating Instances\*\*:

To create an instance (object) of a class, you can call the class as if it were a function, passing any required arguments to the constructor method (`\_\_init\_\_`).

```python

obj1 = MyClass(1, 2)

obj2 = MyClass(3, 4)

```

3. \*\*Accessing Attributes and Methods\*\*:

You can access attributes and methods of an instance using the dot notation:

```python

print(obj1.attribute1) # Accessing an instance attribute

print(obj1.instance\_method()) # Calling an instance method

```

4. \*\*Class Attributes\*\*:

Class attributes are shared among all instances of a class. They are defined at the class level and are accessed using the class name or an instance of the class.

```python

print(MyClass.class\_variable) # Accessing a class attribute

```

5. \*\*Inheritance\*\*:

Python supports class inheritance, which allows you to create a new class based on an existing class. The new class can inherit attributes and methods from the parent class and can also override or extend them.

```python

class ChildClass(MyClass):

def child\_method(self):

return "Child method called"

child\_obj = ChildClass(5, 6)

print(child\_obj.child\_method())

```

6. \*\*Encapsulation, Polymorphism, and Abstraction\*\*:

Python supports OOP principles like encapsulation (hiding data), polymorphism (using objects interchangeably), and abstraction (simplifying complex systems) through the use of classes and objects.

Python classes are a powerful way to structure your code, promote code reusability, and model real-world concepts in a clean and organized manner. They are a fundamental building block for many Python applications, from simple scripts to complex software systems.