Day 03

Generators

Generator functions are functions that allow us to return multiple times using the yield keyword. This allows us to generate many values over time from a single function. What makes generators so powerful is that unlike other forms of iteration, the values are not all computed upfront so we can suspend our state using the yield keyword and come back to to the function later to continue on. This makes generators a great choice for things like calculating large data sets.

```python
def gensquares(n):
    for num in range(n):
        yield num**2

for x in gensquares(10):
    print(x)
```

Using the next function

Given a generator, you can obtain the next value by calling a special function called next and passing in the generator. Here's an example:

```python
def use_next():
    for x in range(10):
        yield x
gen = use_next()
print(next(gen)) # 0
print(next(gen)) # 1
print(next(gen)) # 2
```

In the example above, you can't call next infinitely many times: eventually you'll get a StopIteration error (since x inside of use_next only has finitely many values).

However, if you iterate through a generator using something like a for loop, the loop will catch the error so that it doesn't break your program:

```python
for val in use_next():
    print(val)
```

We can also do generator comprehension just like list comprehension by wrapping our comprehension in () to write generator functions with more ease. Here's how use_next would look as a generator comprehension:

```python
def use_next():
    return (x for x in range(10))
```

Iterators

To make something an iterable (i.e. something you can iterate over) we call the iter function on it. For example, if we wanted strings to be iterators, we could do:

```python
l = [x for x in range(10)]
```

```python

op = iter(l)
print(next(op)) # 0
print(next(op)) # 1
print(next(op)) # 2
print(next(op)) # 3
print(next(op)) # 4
print(next(op)) # 5
print(next(op)) # 6
print(next(op)) # 7
print(next(op)) # 8
print(next(op)) # 9
print(next(op)) # Iteration Error
```

Enumerate

Sometimes when you iterate through an array, you want access not only to the elements, but also their indices. enumerate exposes both to you. It works by returning a tuple with the (index, value) at each iteration.

```python
list = ["first","second","third"]

# How do we get the indices at each iteration? Enumerate!

for idx, value in enumerate(list):
    print(f"index is {idx} and value is {value}")

# index is 0 and value is first
# index is 1 and value is second
# index is 2 and value is third
```

Enums

Enums (Enumerations) are sets of symbolic names bound to unique, constant values.

```python
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

print(Color.RED)

```

all and any

These are both built in functions that all us to check for a boolean matching in an iterable.

all - returns true if **all** elements are truthy

```python
all([0]) # False
all([0,1]) # False
all([0, "", [1]]) # False
all([1, "a", [1]]) # True
any([0]) # False
any([0,1]) # True
any([0, "", [1]]) # True
```

Closures

A closure in Python refers to a function object that has access to variables in its lexical scope, even when the function is called outside of its scope.

```python
def outer_function(text):
    def inner_function():
        print(text)
    return inner_function

my_closure = outer_function('Hello')
my_closure()
```

In other words, Closures are nothing but a function can be assigned to a variable, By then that variable becomes the function. i.e **Higher Order Functions**

Decorators

Decorators are functions that "decorate," or enhance, other functions. In order to see what this means, let's first review how we can pass functions to other functions. Remember, everything in Python is an object and objects are first class!

```python
def shout():
    return "WHOA!"

def whisper():
    return "Shhh"

def perform_action(func):
    print("Something is happening...")
    return func()
perform_action(shout)
# Something is happening...
# WHOA!
perform_action(whisper)
# Something is happening...
# Shhh
```

We can write the behavior of a decorator like this:

```
def decorate_me():
    print("decorate me...")

# decorate_me()
# decorate me...

def new_decorator(func):
    def wrap_func():
        print("Code before calling func!")
        func()
        print("Code after calling func!")
    return wrap_func

dd = new_decorator(decorate_me)
dd()

# Code before calling func!
# decorate me...
# Code after calling func!
```

Think about i don't want to call the name of the function other than original name, how to deal with it ?

```
def decorate_me():
    print("decorate me...")

# decorate_me()
# decorate me...

def new_decorator(func):
    def wrap_func():
        print("Code before calling func!")
        func()
        print("Code after calling func!")
    return wrap_func

decorate_me = new_decorator(decorate_me)
decorate_me()

# Code before calling func!
# decorate me...
# Code after calling func!
```

Let's now use the decorator syntax to do the same thing! When you use a decorator, the function being used to decorate is prefixed with an `@symbol`. The function you're decorating is then defined below. Take a look:

```python
def new_decorator(func):
    def wrap_func():
        print("Code before calling func!")
        func()
        print("Code after calling func!")
    return wrap_func

@new_decorator
def decorate_me():
    print("decorate me...")

decorate_me()

# Code before calling func!
# decorate me...
# Code after calling func!
```

Note how the code inside of the new_decorator function decorates, or enhances, the code inside of the decorate_me function.

Let's revisit the first example but refactor it to use decorator syntax.

```python
def perform_action(func):
    def wrap_func():
        print("Something is happening...")
        return func()
    return wrap_func

@perform_action
def shout():
    return "WHOA!"

@perform_action
def whisper():
    return "Shhh"
shout()
# Something is happening...
# 'WHOA!'
whisper()
# Something is happening...
# 'Shhh'
```

This code will work just fine, but if we examine the `__name__` or `__doc__` attribute for our function it will not be correct!

```python
shout.__name__
# 'wrap_func'
shout.__doc__
# ' Wrapper function '
```

We can manually fix this, or we can use the wraps decorator from the functools module.

```python
from functools import wraps

def perform_action(func):
    ''' decorator function '''
    @wraps(func)
    def wrap_func():
        ''' Wrapper function '''
        print("Something is happening...")
        return func()
    return wrap_func

@perform_action
def shout():
    ''' People Shouts '''
    return "WHOA!"

@perform_action
def whisper():
    ''' People Whispers '''
    return "Shhh"
shout.__name__
# 'shout'
shout.__doc__
# ' People Shouts '
```

Let's extend the previous example by passing arguments to the decorated functions and also alter the data within the decorator function. This is a common scenario where the decorator needs to accept arguments and potentially modify them or perform additional actions based on those arguments.

```python
from functools import wraps

def perform_action(func):
    ''' Decorator function '''
    @wraps(func)
    def wrap_func(*args, **kwargs):
        ''' Wrapper function '''

        # Altering or processing arguments if necessary
        # Whatever the arguments passed converting to upper case
        new_args = [arg.upper() for arg in args]

        # Calling the function with new arguments that are altered
        result = func(*new_args, **kwargs)
        print("Something is happening...")
        # Altering the result if necessary
        return result + '!!!'
    return wrap_func

@perform_action
```

```python
def shout(message):
    ''' People Shouts '''
    return f"WHOA, {message}"

@perform_action
def whisper(message):
    ''' People Whispers '''
    return f"Shhh, {message}"

# Testing the decorated functions
print(shout("hello"))
print(whisper("be quiet"))

# Something is happening...
# WHOA, HELLO!!!
# Something is happening...
# Shhh, BE QUIET!!!

```

Map, Filter and Reduce

Map Function

The `map` function applies a given function to each item of an iterable (like a list or tuple) and returns an iterator. It's often used for transforming data. Here's the syntax:

```python
map(function, iterable, ...)
```

Example: Suppose we want to square each number in a list.

```python
def square(number):
    return number ** 2

numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)

# Convert the map object to a list for display
print(list(squared_numbers))
```

Filter Function

The `filter` function constructs an iterator from elements of an iterable for which a function returns true. Essentially, it filters out the elements of an iterable that don't satisfy a certain condition.

```python
filter(function, iterable)
```

Example: Filtering out even numbers from a list.

```python
def is_even(number):
    return number % 2 == 0

numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(is_even, numbers)

# Convert the filter object to a list for display
print(list(even_numbers))
```

Reduce Function

The `reduce` function, which is part of the `functools` module, applies a rolling computation to sequential pairs of values in an iterable. The function you pass to `reduce` must accept two arguments, and `reduce` applies this function cumulatively to the items of the iterable, from left to right, so as to reduce the iterable to a single value.

```python
from functools import reduce
reduce(function, iterable[, initializer])
```

Example: Use `reduce` to compute the sum of numbers in a list.

```python
from functools import reduce

def add(x, y):
    return x + y

numbers = [1, 2, 3, 4, 5]
result = reduce(add, numbers)

print(result)
```

This example adds up all numbers in the list. `reduce` starts by applying the `add` function to the first two elements (1 and 2), then applies it to the result (3) and the next element (3), and so on, until the list is reduced to a single value.

Key Points

- **map** is used for applying a transformation function to an iterable.
- **filter** is used to select elements of an iterable that meet a certain condition.
- **reduce** is used for cumulatively applying a binary function to the items of an iterable to reduce them to a single value.

Sorting Techniques in Python

Sorting is a fundamental operation in computer science and Python offers a variety of ways to sort data. Let's explore the different sorting techniques available in Python.

Using the `sorted()` Function

The sorted() function returns a new sorted list from the elements of any iterable.

Syntax,

```
sorted(iterable, key=None, reverse=False)
```

Parameters,

- *iterable*: The sequence to sort (list, tuple, string, etc.).
- *key*: A function that serves as a key for the sort comparison.
- *reverse*: If True, the list elements are sorted as if each comparison were reversed.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]
sorted_numbers = sorted(numbers)
print(sorted_numbers)
# Output: [1, 1, 2, 3, 4, 5, 6, 9]
```

Sorting with a Custom Key

The key parameter allows customization of the sort order.

Example,

```
words = ['banana', 'pie', 'Washington', 'apple']
sorted_words = sorted(words, key=len)
print(sorted_words)
# Output: ['pie', 'apple', 'banana', 'Washington']
```

Sorting can be complex when dealing with lists of dictionaries, tuples, or objects.

```
data = [{'name': 'John', 'age': 25}, {'name': 'Jane', 'age': 22}]
sorted_data = sorted(data, key=lambda x: x['age'])
print(sorted_data)  # Sorts data by age
```

Reverse Sorting

You can reverse the sorting order with `reverse=True`.

Example,

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]

sorted_numbers_desc = sorted(numbers, reverse=True)
print(sorted_numbers_desc)
# Output: [9, 6, 5, 4, 3, 2, 1, 1]
```

Understanding `__name__` Significance

In Python, `__name__` is a special built-in variable which plays a crucial role, especially when you are writing and importing modules. Understanding its significance and use cases is essential for advanced Python programming.

What is `__name__` ?

When a Python script runs, the interpreter assigns values to certain special variables. `__name__` is one of these special variables. Its value depends on how the containing script is being executed.

1. When the script is the main program: If the script is being run as the main program, the interpreter sets `__name__` to `__main__`
2. When the script is imported as a module: If the script is being imported as a module into another script, the interpreter sets `__name__` to the name of the script/module.

Why is `__name__` Important ?

The primary use of `__name__` is to determine whether a script is being run standalone or being imported elsewhere. This allows a script to change its behavior based on its context of use. It’s especially useful for running tests, executing initialization code, and providing modules with an entry point.

Example,

Let’s consider two files: `main_script.py` and `imported_module.py`.

file *imported_module.py*

```python
def function_from_module():
    print("Function inside the imported module.")


if __name__ == "__main__":
    print("This script is being run directly")
else:
    print("This script is imported")

```

file *main_script.py*

```python
import imported_module

def function_in_main():
    print("Function inside the main script.")

# First, Run the script with out __name__ module
# if __name__ == "__main__":
#     function_in_main()
#     imported_module.function_from_module()

```

```python
if __name__ == "__main__":
    function_in_main()
    imported_module.function_from_module()

```

Advanced Use Cases of `__name__`

Testing Code: When writing modules, you can place your test code under this `if __name__ == "__main__":` check. This allows you to test the module as a standalone script but avoids running tests when the module is imported.

Making Modules Executable: Sometimes you want a module to be able to act as either a reusable module or as a standalone script. `__name__` allows you to create a module that can do something useful when run on its own, like run a test suite or a demonstration.

Program Entry Point: In larger Python applications, particularly web applications, the `if __name__ == "__main__":` check is used to control the execution of the application. For example, in a Flask web application, this line is used to start the development server.

Exceptional Handling with Python

Exception handling in Python is a robust mechanism to handle runtime errors. Understanding different types of errors and how to manage them is essential for writing robust and fault-tolerant Python programs.

What is Exception Handling ?

Exception handling allows a programmer to respond to unexpected situations that can arise during the execution of a program. In Python, exceptions are special objects representing errors.

Basic Exception Handling: `try`, `except`, `else`, and `finally`

- `try` block: Code that might cause an exception is placed inside a try block.
- `except` block: If an error occurs within the try block, the flow jumps to the except block.
- `else` block (optional): Executed if no exceptions occur within the try block.
- `finally` block (optional): Always executed, regardless of whether an exception occurred.

```python
try:
    # Code that might throw an exception
    result = 10 / 0
except ZeroDivisionError:
    # Code to handle the exception
    print("Divided by zero!")
else:
    # Code to execute if no exceptions
    print("Division successful")
finally:
    # Code that always executes
    print("Execution complete")
```

Types of Errors

Errors in Python can be broadly categorized into two types:

1. **Syntax Errors**: Errors detected by Python as it parses the code. For example, missing colons, incorrect indentation, etc. These cannot be handled by `try`/`except` blocks as they occur before the code is executed.
2. **Exceptions**: Errors detected during execution. Python has numerous built-in exceptions (like `ValueError`, `TypeError`, `IndexError`, `KeyError`, etc.) and also allows creation of custom exceptions.

Common Built-in Exceptions

- **ZeroDivisionError**: Occurs when dividing by zero.
- **IndexError**: Occurs when accessing an index out of range in a sequence.
- **KeyError**: Occurs when a dictionary key is not found.
- **ValueError**: Occurs when a function receives an argument of the correct type but an inappropriate value.
- **TypeError**: Occurs when an operation is performed on an object of an inappropriate type.
- **FileNotFoundError**: Occurs when a file or directory is requested but doesn't exist.
- **ImportError**: Occurs when an import statement fails.

Custom Exceptions

Python allows defining custom exceptions by subclassing from built-in exceptions.

```python
class CustomError(Exception):
    """Base class for custom exceptions"""
    pass

try:
    raise CustomError("An error occurred")
except CustomError as e:
    print(f"Caught custom exception: {e}")

```

Exception handling Best Practices

1. **Specificity**: Catch specific exceptions instead of using a blanket except: clause. This avoids masking other bugs.
2. **Logging**: Log detailed information about the exception.
3. **Clean Resources**: Use finally or context managers to ensure resources are released even if an error occurs.
4. **Raising Exceptions**: Sometimes it's appropriate to catch an exception but re-raise it for upstream handling.

Datetime Module

The `datetime` module in Python is essential for dealing with dates and times. It offers various classes for manipulating dates, times, and time intervals. Understanding these classes and their methods allows you to perform complex date and time calculations with ease.

Key Classes in the `datetime` Module

1. datetime: A combination of a date and a time.
2. date: Represents a date, independent of time.
3. time: Represents a time, independent of a date.
4. timedelta: Represents the difference between two dates or times.

Basic Operations

Getting Current Date and Time

```python
from datetime import datetime

now = datetime.now()
print("Current date and time:", now)

today = datetime.today()
print("Today's date and time:", today)
```

Creating Specific Date and Time

```python
from datetime import datetime

specific_datetime = datetime(2024, 3, 14, 15, 30)  # Year, Month, Day, Hour, Minute
print("Specific date and time:", specific_datetime)
```

Date Operations

```python
from datetime import date

# Create specific dates
date1 = date(2024, 1, 1)
date2 = date(2023, 12, 31)

# Calculate the difference between dates
diff = date1 - date2
print("Difference between dates:", diff)  # Returns a timedelta object
```

Time Operations

Time operations are usually performed by combining time objects with datetime objects or using timedelta objects for calculations.

Imagine we want to schedule an event that starts at a particular time today and lasts for a specific duration. We'll use datetime to get today's date, time to set the event's start time, and timedelta to calculate the event's end time.

```python
from datetime import datetime, time, timedelta

# Get today's date
today = datetime.now().date()

# Define start time of the event (e.g., 14:00 hours or 2 PM)
start_time = time(14, 0)  # 14:00 hours

# Combine today's date and start time
event_start = datetime.combine(today, start_time)
print("Event Start:", event_start)

# Duration of the event (e.g., 1 hour and 30 minutes)
duration = timedelta(hours=1, minutes=30)

# Calculate the event end time
event_end = event_start + duration
print("Event End:", event_end)

print(duration)
```

Extracting Information from datetime

```python
now = datetime.now()

print("Year:", now.year)
print("Month:", now.month)
print("Day:", now.day)
print("Hour:", now.hour)
print("Minute:", now.minute)
print("Second:", now.second)
```

timedelta: Working with Time Differences

The `timedelta` class is used for calculating differences in dates and also for date manipulations in Python.

```python
from datetime import timedelta

# Create a timedelta object
delta = timedelta(days=5, hours=3, minutes=10)

```

```python
print("Delta:", delta)

# Date calculations
future_date = now + delta
print("Future date:", future_date)

# Subtracting dates
past_date = now - delta
print("Past date:", past_date)
```

Formatting Dates and Times

You can format dates and times using strftime (string format time) method.

```python
formatted_date = now.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted date and time:", formatted_date)

# Example: Output in the format 'Mar 14, 2024'
print(now.strftime("%b %d, %Y"))

```

Parsing Dates from Strings

To convert strings to `datetime` objects, use `strptime` (string parse time).

```python
date_string = "2024-03-14 15:30"
parsed_date = datetime.strptime(date_string, "%Y-%m-%d %H:%M")
print("Parsed date:", parsed_date)
```

Timezones in datetime

Handling timezones is a bit more complex. Python's built-in support for timezones is limited, so it's common to use third-party libraries like pytz for comprehensive timezone support.

To Install pytz

```
pip install -U pytz --no-cache
```

Example,

```python
from datetime import datetime
import pytz

utc_now = datetime.now(pytz.utc)
print("UTC Time:", utc_now)

# Convert to a different timezone
indian_time = utc_now.astimezone(pytz.timezone('Asia/kolkota'))
print("Asia/kolkota Time:", indian_time)
```