Procedure Oriented Programming

1. Recall from the previous section that strings are immutable—they can't be changed once they have been created. Most string methods that alter a string, like . methods actually return copies of the original string with the appropriate modifications that can be stored as variable. The original variable remains the same throughout the script.
2. Variables that are created outside of a function (as in all of the examples above) are known as global variables. Global variables can be used by everyone, both inside of functions and outside. If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value. To create a global variable inside a function, you can use the global keyword.

```
def myfunc():
  global x
x= 'fantastic'
myfunc()
print('Python is'  + x)
```

3. The // operator first divides the number on the left by the number on the right and then rounds down to an integer. You can raise a number to a power using the ** operator. The % operator, or the modulus, returns the remainder of dividing the left operand by the right operand:

```
>>> 5 % 3
        2
>>> 20 % 7
        6
```

4. You can use round() to round a number to the nearest integer. All rounding is simple except for ties. Python 3 rounds numbers according to a strategy called rounding ties to even. A tie is any number whose last digit is a five. 2.5 and 3.1415 are ties, but 1.37 is not. When you round ties to even, you first look at the digit one decimal place to the left of the last digit in the tie. If that digit is even, you round down. If the digit is odd, you round up. That's why 2.5 rounds down to 2 and 3.5 rounds up to 4. You can round a number to a given number of decimal places by passing a second argument to round():

```
>>> round(3.14159, 3)
        3.142
>>> round(2.71828, 2)
        2.72
```

5. You can also use the pow() function. pow() takes two arguments. The first is the base, that is the number to be raised to a power, and the second argument is the exponent. For example, the following uses pow() to raise 2 to the exponent 3:

```
>>> pow(2, 3)
        8
```

The pow() function accepts an optional third argument that computes the first number raised to the power of the second number and then takes the modulo with respect to the third number. In other words, pow(x, y, z) is equivalent to (x ** y) % z. Here's an example with x = 2, y = 3, and z = 2:

```
>>> pow(2, 3, 2)
        0
```

Floating-point numbers have an .is_integer() method that returns True if the number is integral—meaning it has no fractional part—and returns False otherwise:

```
>>> num = 2.5
>>> num.is_integer()
False
```

6. To format the value of n to two decimal places, replace the contents of the curly braces in the f-string with {n:.2f}:

```
>>> n = 7.125
>>> f"The value of n is {n:.2f}"
'The value of n is 7.12'
```

The colon (:) after the variable n indicates that everything after it is part of the formatting specification. In this example, the formatting specification is .2f . The .2 in .2f rounds the number to two decimal places, and the f tells to display n as a fixed-point number. This means the number is displayed with exactly two decimal places, even if the original number has fewer decimal places.

The % option should always go at the end of your formatting specification, and you can't mix it with the f option. For example, .1% displays a number as a percentage with exactly one decimal place:

```
>>> ratio = 0.9
>>> f"Over {ratio:.1%} of Pythonistas say \'Real Python rocks!'"
"Over 90.0% of Pythonistas say ' Real Python rocks!'"
```

Thousands representation:

```
>>>x=1500000
>>>f'${x:,.2f}'
'$1,500,000.00'
```

7. To create a complex number in Python, you simply write the real part, followed by a plus sign and the imaginary part with the letter j at the end: i.e n= 1+2j

Imaginary numbers come with two properties, .real and .imag, that return the real and imaginary component of the number, respectively:

```
>>> n.real
1.0
>>> n.imag
2.0
```

Notice that Python returns both the real and imaginary components as floats, even though they were specified as integers.

Complex numbers also have a .conjugate() method that returns the complex conjugate of the number: >>>

```
>>> n.conjugate()
(1-2j)
```

Note: The .real and .imag properties don't need parentheses after them like the method .conjugate() does. The .conjugate() method is a function that performs an action on the complex number. .real and .imag don't perform any action, they just return some information about the number. The distinction between methods and properties is a part of object-oriented programming. Built-in functions are general-purpose and can be used on a wide range of data types (classes) while Methods are class-specific and operate on objects of that class. They are accessed using the dot notation and are defined within the class.

8. UDF:

```
def multiply(x, y): # Function signature
product = x * y # Function body
return product
```

The second line of code is called a return statement. It starts with the return keyword and is followed by the variable product. When Python reaches the return statement, it stops running the function and returns the value of product

Recursion is a programming technique in which a function calls itself to solve a problem. In Python, you can create recursive functions just like any other functions. Recursive functions typically have two parts:

Base Case: A condition that defines when the recursion should stop. When the base case is met, the function returns a value without making further recursive calls.
Recursive Case:The part of the function where it calls itself with modified arguments to make progress towards the base case.
Here's an example of a simple recursive function to calculate the factorial of a non-negative integer:

```python
def factorial(n):
    # Base case: If n is 0 or 1, return 1 (factorial of 0 and 1 is 1)
    if n == 0 or n == 1:
        return 1
    else:
        # Recursive case: Call the factorial function with n-1
        return n * factorial(n - 1)
# Example usage:
result = factorial(5)
print(result)  # Output: 120 (5! = 5 * 4 * 3 * 2 * 1 = 120)
```

In this example, `factorial(n)` is defined recursively. When `n` is 0 or 1 (the base case), it returns 1. Otherwise, it makes a recursive call to `factorial(n - 1)` and multiplies the result by `n`.
Keep in mind the following principles when working with recursive functions:

1. Ensure that you have a base case to prevent infinite recursion.
2. Make sure that each recursive call moves closer to the base case.
3. Recursive functions can be less efficient than iterative solutions for some problems, so consider the trade-offs when choosing recursion.

Recursion is a powerful technique and can be used to solve a wide range of problems, such as tree and graph traversal, divide-and-conquer algorithms, and more. However, it's important to use recursion judiciously and understand when it's an appropriate choice for a particular problem.

Lambda functions, also known as anonymous functions or lambda expressions, are a way to create small, inline functions without the need for a formal function definition. They are particularly useful for simple operations and can take any number of arguments but can only consist of a single expression.

The syntax of a lambda function is as follows:

```python
lambda arguments: expression
```

Here's an example of a lambda function that takes three arguments and returns their sum:

```python
x = lambda a, b, c: a + b + c
result = x(5, 6, 2)
print(result)  # Output: 13
```

In this example, `x` is a lambda function that takes three arguments (`a`, `b`, and `c`) and returns their sum. When we call `x(5, 6, 2)`, it calculates `5 + 6 + 2` and returns `13`.

Lambda functions are often used in situations where you need a simple, short function for a specific task and don't want to define a separate named function using the `def` keyword. They are commonly used in functions like `map()`, `filter()`, and `sorted()` that accept functions as arguments.

9. Consider the following script. What do you think the output is?

```
total = 0
    def add_to_total(n):
    total = total + n
    add_to_total(5)
    print(total)
```

Error!

Now try this:

```
total = 0
def add_to_total(n):
global total
total = total + n
add_to_total(5)
print(total)
```

This time, you get the expected output 5.

Why's that? The line global total tells Python to look in the global scope for the name total. That way, the line total = total + n does not create a new local variable.

10. In Python, strings are ordered lexicographically, which is a fancy way to say they are ordered as they would appear in a dictionary. So you can think of "a" < "b" as asking whether or not the letter a comes before the letter b in the dictionary. Lexicographic ordering extends to strings with two or more characters by looking at each component letter of the string:

```
>>> "apple" < "astronaut"
True
>>> "beauty" > "truth"
False
```

3. **`break` Statement**: The **`break`** statement is used to exit a loop prematurely, regardless of whether the loop condition is met.

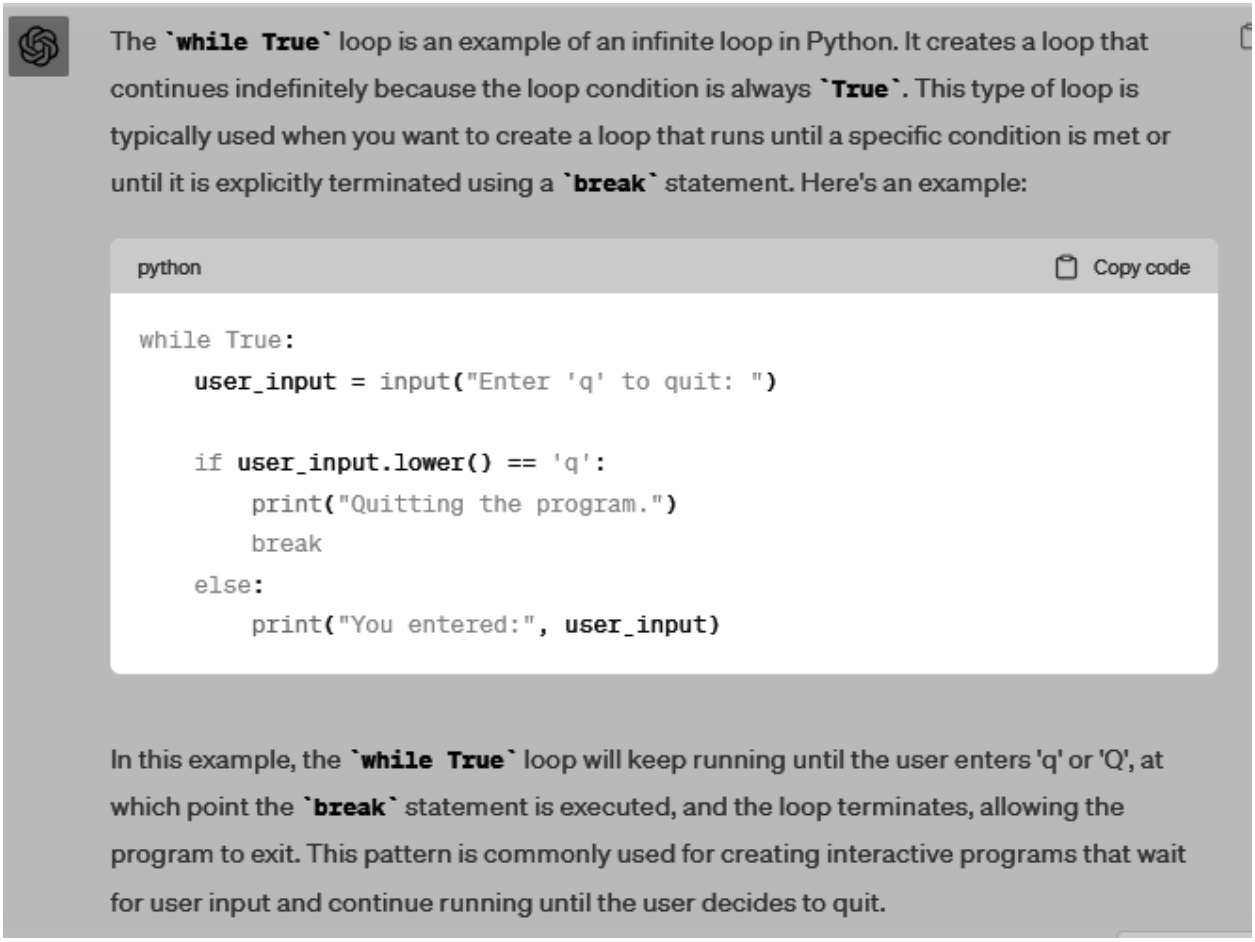
python Copy code

```python
while True:
    if condition:
        break
```

4. **`continue` Statement**: The **`continue`** statement is used to skip the current iteration of a loop and continue with the next iteration.

python Copy code

```python
for item in sequence:
    if condition:
        continue
    # Code here will be skipped for specific items
```



The `while True` loop is an example of an infinite loop in Python. It creates a loop that continues indefinitely because the loop condition is always `True`. This type of loop is typically used when you want to create a loop that runs until a specific condition is met or until it is explicitly terminated using a `break` statement. Here's an example:

python Copy code

```
while True:
    user_input = input("Enter 'q' to quit: ")

    if user_input.lower() == 'q':
        print("Quitting the program.")
        break
    else:
        print("You entered:", user_input)
```

In this example, the `while True` loop will keep running until the user enters 'q' or 'Q', at which point the `break` statement is executed, and the loop terminates, allowing the program to exit. This pattern is commonly used for creating interactive programs that wait for user input and continue running until the user decides to quit.

To summarize, the break keyword is used to stop a loop if a certain condition is met, and the continue keyword is used to skip an iteration of a loop when a certain condition is met.

In Python, escape characters are special sequences of characters that are used within strings to represent characters that cannot be easily typed or displayed directly. Escape characters are preceded by a backslash (`\`) and are used to insert special characters or control codes into strings:
`\\`: Backslash
- Used to represent a literal backslash character.
- Example: `"This is a backslash: \\"`

11. When using a while loop in Python, it's crucial to ensure that the loop control variable (often denoted as i) is properly incremented or adjusted inside the loop. Failure to do so can result in an infinite loop, causing your program to hang or become unresponsive.

Here's a simple example of a while loop that increments i:

```
''
i = 0  # Initialize i
while i < 5:
    print(i)
    i += 1  # Increment i by 1 in each iteration
```

In this example, i is incremented by 1 in each iteration of the loop, preventing it from running infinitely. The loop will execute until i is no longer less than 5.
Always be mindful of loop control variables and make sure they are properly updated within the loop to avoid unintended infinite loops.

In Python, the `try` and `except` blocks are used for error handling and exception handling. They allow you to gracefully handle exceptions (errors) that may occur during the execution of your code. Here's the basic syntax:

python Copy code

```python
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
```

- The `try` block contains the code that you want to monitor for exceptions.
- If an exception of type `ExceptionType` (or its subclass) occurs inside the `try` block, the code inside the `except` block is executed to handle the exception.

Here are some common scenarios for using `try` and `except` blocks:

1. **Catching Specific Exceptions:**

   You can catch specific types of exceptions and handle them differently:

python Copy code

```python
try:
    result = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Division by zero is not allowed.")
```

Rege

2. **Handling Multiple Exceptions:**

   You can handle multiple types of exceptions in the same `` `try` `` block using multiple `` `except` `` blocks:

```python
try:
    result = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Division by zero is not allowed.")
except ValueError:
    print("Invalid input value.")
```

3. **Using a Generic Exception:**

   You can use a generic `` `except` `` block to catch any exception type:

```python
try:
    result = 10 / 0  # This will raise a ZeroDivisionError
except Exception as e:
    print(f"An error occurred: {e}")
```

   However, it's generally recommended to catch specific exceptions whenever possible to handle them appropriately.

4. **Cleanup with `finally`:**

   You can use a `finally` block to define code that will be executed regardless of whether an exception occurred or not. It's often used for cleanup tasks:

```python
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
finally:
    # Cleanup code (executed regardless of whether an exception occurred)
```

   Here's an example that combines various aspects of exception handling:

```python
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ZeroDivisionError:
    print("Division by zero is not allowed.")
except ValueError:
    print("Invalid input. Please enter a valid number.")
else:
    print(f"Result: {result}")
finally:
    print("Execution completed.")
```

   In this example, the program attempts to get user input, perform a division, and handle various types of exceptions. The `else` block is executed if no exceptions occur, and the `finally` block is always executed, regardless of whether an exception occurred or

12. The following code produces a random integer between 1 and 10:

```
>>> random.randint(1, 10)
9
```

13. You can also use the tuple() built-in to create a tuple from another sequence type, such as a string:

```
>> tuple("Python")
        ('P', 'y', 't', 'h', 'o', 'n')
```

Tuples and strings have a lot in common. Both are sequence types with a finite length, support indexing and slicing, are immutable, and can be iterated over in a loop. The main difference between strings and tuples is that the elements of tuples can be any kind of value you like, whereas strings can only contain characters.

Aside from list literals, you can also use the list() built-in to create a new list object from any other sequence. For instance, the tuple (1, 2, 3) can be passed to list() to create the list [1, 2, 3]:

```
>>> list((1, 2, 3))
    [1, 2, 3]
```

You can create a list from a string of a comma-separated list of items using the string object's .split() method:

```
>>> groceries = "eggs, milk, cheese"
>>> grocery_list = groceries.split(", ")
>>> grocery_list
['eggs', 'milk', 'cheese']
```

The string argument passed to `.split()` is called the separator. By changing the separator you can split strings into lists in numerous ways:

```
>>> # Split string on semi-colons
>>> "a;b;c".split(";")
['a', 'b', 'c']

>>> # Split string on spaces
>>> "The quick brown fox".split(" ")
['The', 'quick', 'brown', 'fox']

>>> # Split string on multiple characters
>>> "abbaabba".split("ba")
['ab', 'ab', '']
```

List Method	Description
`.insert(i, x)`	Insert the value `x` at index `i`
`.append(x)`	Insert the value `x` at the end of the list

List Method	Description
`.extend(iterable)`	Insert all the values of `iterable` at the end of the list, in order
`.pop(i)`	Remove and return the element at index `i`

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created. But we can convert them to list, perform changings and then turn back to tuple.
You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference (pointer) to list1, and changes made in list1 will automatically also be made in list2.
There are ways to make a copy, one way is to use the built-in List method copy().
Make a copy of a list with the copy() method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

14. sort() has an option parameter called key that can be used to adjust how the list gets sorted. The key parameter accepts a function, and the list is sorted based on the return value of that function. For example, to sort a list of strings by the length of each string, you can pass the len function to key:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors.sort(key=len)
                >>> colors ['red', 'blue', 'green', 'yellow']
```

15. You can create an empty dictionary using either a literal or dict():

```
>>> {}
        {}
>>> dict()
        {}
```

Now we have created dict. We can add values in it as we need.

del capitals['Texas'] will delete key from

Whether a method changes the actual variable depends on whether the variable is pointing to an immutable or mutable object and how the method is implemented. For mutable objects, like lists, dictionaries, and sets, you should be aware that methods can modify them directly. To avoid unintentional side effects, you may need to make a copy of the object if you want to keep the original data intact while performing operations. For immutable objects, like numbers and strings, operations or methods create new objects and update the variable to reference the new object. The original object remains unchanged. Understanding the mutability of objects in Python is crucial for writing code that behaves as expected and for avoiding unexpected side effects when working with variables and data structures.

1. Lists:
- Ordered: Items are stored in a specific order, and you can access them by their index.
- Changeable: You can add, remove, and modify elements in a list.
- Allows duplicate members: You can have multiple elements with the same value in a list.

2. Tuples:
- Ordered: Like lists, items are stored in a specific order, and you can access them by their index.
- Unchangeable: Once you create a tuple, you cannot add, remove, or modify its elements.
- Allows duplicate members: Tuples can contain duplicate values.

3. Sets:
- Unordered: Set items have no specific order, so you cannot access them by index.
- Unchangeable*: While the items themselves are unchangeable, you can add and remove items from a set.
- No duplicate members: Sets automatically remove duplicate values, so each element is unique.

4. Dictionaries:
- Ordered** (Python 3.7+): Dictionaries maintain the insertion order of their items.
- Changeable: You can add, remove, and modify key-value pairs in a dictionary.
- No duplicate members: Dictionary keys must be unique, but values can be duplicated.

Object Oriented Programming

Classes are used to create user-defined data structures. Classes also have special functions, called methods, that define behaviors and actions that an object created from the class can perform with its data. Classes are used to create user-defined data structures. Classes also have special functions, called **methods**, that define behaviors and actions that an object created from the class can perform with its data. i.e.

```
class dog:
    pass
```

In Object-Oriented Programming (OOP) with Python, the pass statement is used as a placeholder when you define a class, method, or function that doesn't contain any code yet but is required syntactically. It serves as a way to indicate that you intend to implement the class or method later but don't want to provide any specific implementation at the moment.

While the class is the blueprint, an **instance** is an **object** built from a class that contains real data. An instance of the Dog class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

To define the properties, or **instance attributes**, that all Dog objects must have, you need to define a special method called .__init__(). This method is run every time a new Dog object is created and tells Python what the initial state—that is, the initial values of the object's properties—of the object should be. The first positional argument of .__init__() is always a variable that references the class instance. This

variable is almost universally named self. After the self argument, you can specify any other arguments required to create an instance of the class. The following updated definition of the Dog class shows how to write an .__init__() method that creates two instance attributes: .name and .age:

```
class Dog:
   def __init__(self, name, age):
   self.name = name
   self.age = age
```

In the body of the .__init__() method, there are two statements using the self variable. The first line, self.name = name, creates an instance attribute called name and assigns to it the value of the name variable that was passed to the .__init__() method. The second line creates an instance attribute called age and assigns to it the value of the age argument.

This might look kind of strange. The self-variable is referring to an instance of the Dog class, but we haven't actually created an instance yet. It is a place holder that is used to build the blueprint. Remember, the class is used to define the Dog data structure. It does not actually create any instances of individual dogs with specific names and ages.

While instance attributes are specific to each object, **class attributes** are the same for all instances—which in this case is all dogs. In the next example, a class attribute called species is created and assigned the value "Canis familiaris":

```
class Dog:
# Class Attribute
species = "Canis familiaris"
def __init__(self, name, age):
self.name = name
self.age = age
```

Class attributes are defined directly underneath the first line of the class and outside of any method definition. They must be assigned a value because they are created on a class instance without arguments 283 10.2. Instantiate an Object to determine what their initial value should be

We feed instances as follow:

```
class Dog:
   species = "Canis familiaris"
   def __init__(self, name, age):
   self.name = name
   self.age = age
>>> buddy = Dog("Buddy", 9)
>>> miles = Dog("Miles", 4)
#attributes are accessed as follow:
>>> buddy.age
9
>>> miles.name
'Miles'
>>> miles.age
4
>>>buddy.species
'Canis familiaris'
#Instance methods are functions defined inside of a class.
```

```
    def description(self):
        return f"{self.name} is {self.age} years old"
    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
    #Accessing these methods
    >>>miles.speak("Bow Wow")
    'Miles says Bow Wow'
```

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called child classes, and the classes that child classes are derived from are called parent classes. Child classes can override and extend the attributes and methods of parent classes. In other words, child classes inherit all of the parent's attributes and methods but can also specify different attributes and methods that are unique to themselves, or even redefine methods from their parent class.

Remember, to create a child class, you create new class with its own name and then put the name of the parent class in parentheses.

Example:

```
class rectangle:

    def __init__(self,len,wid):
        self.len=len
        self.wid=wid
    def area(self):
        print(f'{self.len*self.wid}')
class square(rectangle):
    def __init__(self,sidelen):
        self.sidelen=sidelen
        super(). __init__(len=sidelen,wid=sidelen)
Square=square(4)
Square.area()
>>>16
```

Modify Properties:

You can change the value of an object's property by assigning a new value to it, as shown in your example:

```python
Square.sidelen = 40
```

This sets the `sidelen` property of object `Square` to the value 40.

You can delete a property of an object using the `del` keyword, as demonstrated in your example:

```python
del Square.sidelen
```

This removes the `sidelen` property from object `square`.

You can delete an entire object, freeing up memory, using the `del` keyword followed by the object's name:

```python
del Square
```

This deletes the entire `Square` object, and you won't be able to access it afterward.

A **module** is a file containing Python code that can be re-used in other Python code files. Creating Modules: Open IDLE and start a new script window by selecting File New File or by pressing Ctrl + N . In the script window, define a function add() that returns the sum of its two parameters:

```
# adder.py
def add(x, y):
    return x + y
def double(x):
    return x+x
```

Select File Save or press Ctrl + S to save the file as adder.py in a new directory called myproject/ somewhere on your computer. adder.py is a Python module! It's not a complete program, but not all modules need to be.

Importing:

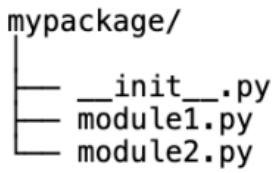
When you import one module into another, the contents of the imported module become available in the other. The module with the import statement is called the **calling module.** A **namespace** is a collection of names, such as variable names, function names, and class names. Every Python module has its own namespace. Variables, functions, and classes in a module can be accessed from within the same module by just typing their name. That's how you've been doing it throughout this book so far. However, this doesn't work for imported modules. To access a name in an imported module from the calling module, type the imported module's name followed by a dot (.) and the name you want to use:

```
import adder
value1 = adder.add(2, 2)
value2=adder.double(2)
print(value1,value2)
>> 4  4
```

Import statement variations:

Import Statement	Result
`import <module>`	Import all of `<module>`'s namespace into the name `<module>`. Import module names can be accessed from the calling module with `<module>.<name>`.
`import <module> as <other_name>`	Import all of `<modules>`'s namespace into the name `<other_name>`. Import module names can be accessed from the calling module with `<other_name>.<name>`.
`from <module> import <name1>, <name2>, ...`	Import only the names `<name1>`, `<name2>`, etc, from `<module>`. The names are added to the calling modules's local namespace and can be accessed directly.

A **package** is a folder that contains one or more Python modules. It must also contain a special module called __init__.py. Here is an example of a package so that you can see this structure:



Note:The __init__.py module doesn't need to contain any code! It only needs to exist so that Python recognizes the mypackage/ folder as a Python package.

Importing:

import *<package_name>.<module_name>*

```
import mypackage.module1
import mypackage.module2  # <-- Add this line

# Leave the below code unchanged
mypackage.module1.greet("Pythonista")
mypackage.module2.depart("Pythonista")
```

Now when you save and run `main.py`, both `greet()` and `depart()` get called:

```
Hello, Pythonista!
Goodbye, Pythonista!
```

Variations:

1. import <package>

2. import <package> as <other_name>

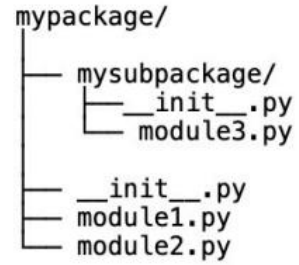
3. from <package> import <module>

4. from <package> import <module> as <other_name>

5. from mypackage import module1, module2,…

Subpakages:

A package is just a folder containing one or more Python modules, one of which must be names __init__.py, so it's entirely possible to have the following package structure:



A package nested inside of another package is called a **subpackage**. For example, the mysubpackage

folder is a subpackage of mypackage because it contains an __init__.py module, as well as a second module named module3.py.

In your `module3.py` file, add the following code:

```
# module3.py

people = ["John", "Paul", "George", "Ringo"]
```

Now open the `main.py` file in your root `packages_examples/` project folder. Remove any existing code and replace it with the following:

```
# main.py

from mypackage.module1 import greet
from mypackage.mysubpackage.module3 import people

for person in people:
    greet(person)
```

>>>Hello, John!
Hello, Paul!
Hello, George!
Hello, Ringo!

The `dir()` function is a built-in Python function that can be used to list all the function names, variable names, and other attributes (such as classes, methods, and modules) defined in a module or object. It returns a list of strings representing the names of these attributes.
Here's how you can use the `dir()` function:
1. To list all attributes in a module:
   ```python
   import module_name
   print(dir(module_name))
   ```
Replace `module_name` with the name of the module you want to inspect. This will list all functions, variables, classes, and other attributes defined in the module.
2. To list all attributes in an object:
   ```python
   object = SomeClass()  # Create an instance of a class or any object
   print(dir(object))
   ```
This will list all attributes and methods associated with the object, including inherited attributes and methods from parent classes.
Keep in mind that the `dir()` function provides a comprehensive list of attributes, including Python's built-in attributes and methods, so the output can be quite extensive. You can use filtering and additional logic to extract specific types of attributes or filter out those you are not interested in.