Python Fundamentals - Task 2

By Menna Jaheen

Functions :

- A function is a block of organized code that is used to perform a single task. They provide better modularity for your application and reuse-ability.
- A function can take `arguments` and `return values`:

```
def say_hello(name):
    print(f'Hello {name}')
```

In the function above : the argument is the "name " , there is no return type but it prints a greeting .

we can use a `keyword argument` *→ arguments prefixed with the names of parameters , order of the arguments doesn't matter but it helps with readability:*

```
def hello(greeting, title, first, last):
    print(f"{greeting} {title}{first} {last}")
```

- When creating a function using the `def` statement, you can specify what the return value should be with a `return` statement. A return statement consists of the following:
  - The `return` keyword.
  - The value or expression that the function should return.

```
def sum_two_numbers(number_1, number_2):
    return number_1 + number_2
```

Modules:

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable.

why modules ?

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead .Such a file is called a *module*

Example :

Create a file called `fibo.py` in your current directory with the following contents:

```
# Fibonacci numbers module
def fib(n):     # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
       print(a, end=' ')
       a, b = b, a+b
    print()

def fib2(n):        # return Fibonacci series up to n
```

```
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
return result
```

Now enter the Python interpreter and import this module with the following command: `import fibo`

NOTE :

This does not add the names of the functions defined in `fibo` directly to the current namespace ; it only adds the module name `fibo` there. Using the module name you can access the functions:

```
fibo.fib(1000)
```

Data Structures :

Lists :

- **A list is an ordered, mutable collection of items.**
- **Characteristics:**
  - Can store items of different data types.
  - Supports indexing and slicing.
  - Elements can be added, removed, or changed.
- **Syntax**: Defined using square brackets `[]`.
- **Example**: `my_list = [1, 2, "apple", 4.5]`

Some of the functions used with lists :

```
list.insert(i, x) #Insert an item at a given position.
list.append(x)   #Add an item to the end of the list.
list.remove(x) #Remove the first item from the list whose value is equal to x.
list.pop([i]) #Remove the item at the given position in the list, and return it.
list.clear() #Remove all items from the list. Equivalent to del a[:].
list.count(x) #Return the number of times x appears in the list.
list.sort(*, key=None, reverse=False) #Sort the items of the list in place
list.reverse() #Reverse the elements of the list in place.
list.copy() #Return a shallow copy of the list. Equivalent to a[:].
```

Dictionaries :

A dictionary is an unordered collection of key-value pairs.

- **Characteristics**:
  - Keys must be unique and immutable.
  - Values can be of any data type and can be duplicated.
  - Fast lookups by key.
- **Syntax**: Defined using curly braces `{}` with key-value pairs separated by colons.
- **Example**: `my_dict = {"name": "Alice", "age": 25, "city": "New York"}`

Some of the functions used with Dictionaries :

- **Accessing a value by key**: Using square brackets `[]` or `get()`

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
print(my_dict["name"])  # Output: Alice
print(my_dict.get("age"))  # Output: 25
```

- **Adding or updating a key-value pair**:

```
my_dict["job"] = "Engineer"
print(my_dict)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York', 'job': 'Enginee
r'}
```

- **Removing a key-value pair**: `pop()`

```
removed_value = my_dict.pop("age")
print(removed_value)  # Output: 25
print(my_dict)  # Output: {'name': 'Alice', 'city': 'New York', 'job': 'Engineer'}
```

- **Getting all keys**: `keys()`

```
keys = my_dict.keys()
print(keys)  # Output: dict_keys(['name', 'city', 'job'])
```

- **Getting all values**: `values()`

```
values = my_dict.values()
print(values)  # Output: dict_values(['Alice', 'New York', 'Engineer'])
```

Tuples :

A tuple is an ordered, immutable collection of items.

- **Characteristics**:
  - Similar to lists but cannot be changed after creation.
  - Supports indexing and slicing.
  - Used for fixed data sequences.
- **Syntax**: Defined using parentheses `()`.
- **Example**: `my_tuple = (1, 2, "apple", 4.5)`

- **Accessing elements by index**:

```
my_tuple = (1, 2, "apple", 4.5)
print(my_tuple[2])  # Output: apple
```

- **Counting occurrences of an element**: `count()`

```
my_tuple = (1, 2, 2, 3)
print(my_tuple.count(2))  # Output: 2
```

- **Finding the index of an element**: `index()`

```
print(my_tuple.index(3))  # Output: 3
```

Sets :

A set is an unordered collection of unique items.

- **Characteristics**:
  - Does not allow duplicate elements.
  - Useful for membership testing and removing duplicates from sequences.
  - Supports set operations like union, intersection, and difference.
- **Syntax**: Defined using curly braces `{}` or the `set()` function.
- **Example**: `my_set = {1, 2, 3, "apple"}`

Some of the functions used with sets :

- **Adding an element**: `add()`

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)  # Output: {1, 2, 3, 4}
```

- **Removing an element**: `remove()` or `discard()`

```
my_set.remove(3)
print(my_set)  # Output: {1, 2, 4}
```

```
my_set.discard(4)
print(my_set)  # Output: {1, 2}
```

- **Performing union with another set**: `union()`

```
another_set = {3, 4, 5}
union_set = my_set.union(another_set)
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

- **Performing intersection with another set**: `intersection()`

```
intersection_set = my_set.intersection(another_set)
print(intersection_set)  # Output: {3, 4}
```

- **Clearing all elements from the set**: `clear()`

```
my_set.clear()
print(my_set)  # Output: set()
```

Error Handling :

In Python, errors are generally categorized into two types: **syntax errors** and **exceptions** (runtime errors). Understanding these error types is essential for writing robust code and effectively handling problems that arise during execution.

Exceptions (Runtime Errors)

- **Definition**: Exceptions are errors that occur during the execution of a program. Unlike syntax errors, exceptions do not prevent the program from starting, but they can cause it to terminate if not properly handled.
- **Categories of Exceptions**:
  - **Built-in Exceptions**: Python comes with a rich set of built-in exceptions to handle various common error conditions.
  - **Custom Exceptions**: Developers can define their own exceptions by inheriting from the `Exception` class.

Common Built-in Exceptions:

Type Error

Raised when an operation or function is applied to an object of inappropriate type.

```
x = 5 + "hello"
```

- **Error Message:** `TypeError: unsupported operand type(s) for +: 'int' and 'str'`

Value Error

Raised when a function receives an argument of the right type but an inappropriate value.

```
x = int("abc")
```

- **Error Message:** `ValueError: invalid literal for int() with base 10: '`

Zero Division Error

Raised when attempting to divide a number by zero.

```
x = 10 / 0
```

- **Error Message:** `ZeroDivisionError: division by zero`

Handling Exceptions:

```
try:
number = int(input("Enter a number: "))
print(1 / number)
except ZeroDivisionError:
print("You can't divide by zero IDIOT!")
except ValueError:
print("Enter only numbers please!")
except Exception:
print("Something went wrong!")
finally:
print("Do some cleanup here")
```