introduction-to-python

March 10, 2024

1 Primitive Data types

- Number
  - int
  - float
  - complex
- String
- Boolean
- None

[ ]:
```
age = 30 # number (int)
pi = 3.14 # number (float)
name = "John" # string
married = False # boolean
address = None # None (equivalent to null, undefine)

print(age,pi,name,married,address)
```

```
30 3.14 John False None
```

2 Composite Data Types

- List
- Dictionary
- Tuple
- Set
- Range

2.1 List

- A list is a mutable and ordered collection of elements in Python.
- It allows duplicate elements and maintains the order of insertion.
- Lists are versatile and can contain elements of different data types.
- Lists are defined by enclosing comma-separated elements within square brackets [ ]

[ ]:
```
favoriteFoods = ['Pizza','MoMo'] # create new list
```

```python
#  Append an item to the list
favoriteFoods.append('Sushi')
print("After appending 'Sushi':", favoriteFoods)

#  Insert an item at a specific position
favoriteFoods.insert(1, 'Burger')
print("After inserting 'Burger' at index 1:", favoriteFoods)

#  Remove an item from the list
favoriteFoods.remove('MoMo')
print("After removing 'MoMo':", favoriteFoods)

#  Pop an item from the list
popped_item = favoriteFoods.pop()
print("Popped item:", popped_item)
print("List after popping:", favoriteFoods)

# Accessing an item by index
print("First item in the list:", favoriteFoods[0])

# Slicing the list
print("Sliced list from index 1 to 3:", favoriteFoods[1:3])

# Length of the list
print("Length of the list:", len(favoriteFoods))
```

```
After appending 'Sushi': ['Pizza', 'MoMo', 'Sushi']
After inserting 'Burger' at index 1: ['Pizza', 'Burger', 'MoMo', 'Sushi']
After removing 'MoMo': ['Pizza', 'Burger', 'Sushi']
Popped item: Sushi
List after popping: ['Pizza', 'Burger']
First item in the list: Pizza
Sliced list from index 1 to 3: ['Burger']
Length of the list: 2
```

2.2 Dictionary

- A dictionary is a mutable and unordered collection of key-value pairs in Python.
- It provides a mapping between keys and values, allowing efficient data retrieval.
- Keys within a dictionary are unique and immutable, while values can be of any data type and mutable.
- Dictionaries are defined by enclosing comma-separated key-value pairs within curly braces { }, with each pair separated by a colon :

[ ]:
```python
# Create a person dictionary
person = {
    'name': 'John Doe',
    'age': 30,
```

```
    'occupation': 'Software Engineer',
    'city': 'New York'
}

# Print the original dictionary
print("Original dictionary:")
print(person)

# Accessing value by key
print("\nAccessing value by key:")
print("Name:", person['name'])
print("Age:", person['age'])

# Adding a new key-value pair
print("\nAdding a new key-value pair:")
person['gender'] = 'Male'
print("After adding 'gender':", person)

# Updating a value
print("\nUpdating a value:")
person['age'] = 32
print("After updating 'age':", person)

# Removing a key-value pair
print("\nRemoving a key-value pair:")
removed_value = person.pop('occupation')
print("Removed value:", removed_value)
print("After removing 'occupation':", person)

# Checking if a key exists
print("\nChecking if a key exists:")
print("'city' exists in person dictionary:", 'city' in person)
print("'email' exists in person dictionary:", 'email' in person)

# Iterating over keys and values
print("\nIterating over keys and values:")
for key, value in person.items():
    print(key + ":", value)

# Getting keys and values as lists
print("\nGetting keys and values as lists:")
keys = list(person.keys())
values = list(person.values())
print("Keys:", keys)
print("Values:", values)

# Length of the dictionary
```

```python
print("\nLength of the dictionary:", len(person))

# Clearing the dictionary
print("\nClearing the dictionary:")
person.clear()
print("Dictionary after clearing:", person)
```

```
Original dictionary:
{'name': 'John Doe', 'age': 30, 'occupation': 'Software Engineer', 'city': 'New
York'}

Accessing value by key:
Name: John Doe
Age: 30

Adding a new key-value pair:
After adding 'gender': {'name': 'John Doe', 'age': 30, 'occupation': 'Software
Engineer', 'city': 'New York', 'gender': 'Male'}

Updating a value:
After updating 'age': {'name': 'John Doe', 'age': 32, 'occupation': 'Software
Engineer', 'city': 'New York', 'gender': 'Male'}

Removing a key-value pair:
Removed value: Software Engineer
After removing 'occupation': {'name': 'John Doe', 'age': 32, 'city': 'New York',
'gender': 'Male'}

Checking if a key exists:
'city' exists in person dictionary: True
'email' exists in person dictionary: False

Iterating over keys and values:
name: John Doe
age: 32
city: New York
gender: Male

Getting keys and values as lists:
Keys: ['name', 'age', 'city', 'gender']
Values: ['John Doe', 32, 'New York', 'Male']

Length of the dictionary: 4

Clearing the dictionary:
Dictionary after clearing: {}
```

2.3 Tuple

- A tuple is an immutable and ordered collection of elements in Python.
- It allows duplicate elements and maintains the order of insertion.
- Tuples are commonly used for representing fixed collections of related data.
- Tuples are defined by enclosing comma-separated elements within parentheses ( ).

[ ]:
```
# Create a tuple representing information about a book
book = ('Python Programming', 'John Smith', 2022, 400)

# Printing the tuple
print("Book tuple:", book)

# Describing properties of the tuple
print("\nProperties of the tuple:")

# Immutable
print("Immutable: Tuples are immutable, meaning their elements cannot be␣
 ↪changed once assigned.")
# Uncomment the following line to see the error generated when trying to change␣
 ↪a tuple element
# book[0] = 'New Title'

# Ordered
print("Ordered: Tuples maintain the order of their elements.")
print("First element:", book[0])

# Heterogeneous
print("Heterogeneous: Tuples can contain elements of different data types.")
print("Author:", book[1])
print("Year:", book[2])

# Allows duplicates
print("Allows duplicates: Tuples can contain duplicate elements.")
print("Number of pages:", book[3])

# Size is fixed
print("Size is fixed: Once created, the size of a tuple cannot be changed.")
print("Length of the tuple:", len(book))
```

```
Book tuple: ('Python Programming', 'John Smith', 2022, 400)

Properties of the tuple:
Immutable: Tuples are immutable, meaning their elements cannot be changed once
assigned.
Ordered: Tuples maintain the order of their elements.
First element: Python Programming
Heterogeneous: Tuples can contain elements of different data types.
```

```
Author: John Smith
Year: 2022
Allows duplicates: Tuples can contain duplicate elements.
Number of pages: 400
Size is fixed: Once created, the size of a tuple cannot be changed.
Length of the tuple: 4
```

2.4 Set

- A set is a mutable and unordered collection of unique elements in Python.
- It does not allow duplicate elements and is primarily used for mathematical operations - like union, intersection, and difference.
- Sets are defined by enclosing comma-separated elements within curly braces { }.

[ ]:
```python
# Create a set representing unique tags for a blog post
tags = {'python', 'programming', 'tutorial', 'python'}

# Printing the set
print("Tags set:", tags)

# Describing properties of the set
print("\nProperties of the set:")

# Unordered
print("Unordered: Sets do not maintain the order of their elements.")
# Uncomment the following line to see the elements printed in different order
# print("Elements in set:", tags)

# Unique elements
print("Unique elements: Sets contain only unique elements.")
print("Number of unique tags:", len(tags))

# Mutable
print("Mutable: Sets are mutable, meaning elements can be added or removed.")
tags.add('web development')
print("After adding 'web development':", tags)
tags.remove('tutorial')
print("After removing 'tutorial':", tags)

# Size can change
print("Size can change: Sets can grow or shrink dynamically.")
print("Length of the set:", len(tags))
```

```
Tags set: {'python', 'programming', 'tutorial'}

Properties of the set:
Unordered: Sets do not maintain the order of their elements.
Unique elements: Sets contain only unique elements.
```

```
Number of unique tags: 3
Mutable: Sets are mutable, meaning elements can be added or removed.
After adding 'web development': {'python', 'programming', 'web development',
'tutorial'}
After removing 'tutorial': {'python', 'programming', 'web development'}
Size can change: Sets can grow or shrink dynamically.
Length of the set: 3
```

3 Control Structures (Conditional Statements)

- if
- if-else
- if-elif-else (if-elseif-else)
- Nested if statements
- Ternary conditional operator

Note: Python doesn't have switch

[ ]:
```python
time = 10
# if
if time<12:
    print('Good Morning')

# if-else

day = 'Saturday'
if day=='Saturday':
    print('Its holiday')
else:
    print('Its weekday')

# if-elif-else
fuel_type='petrol'

if fuel_type == 'electric':
    print("Electric vehicle")
elif fuel_type == 'petrol':
    print("petrol-powered vehicle")
elif fuel_type == 'diesel':
    print("Diesel-powered vehicle")
else:
    print("Unknown fuel type")

# ternary operator
x = 10
y = 20
max_value = x if x > y else y
print(max_value)  # Output: 20
```

```
Good Morning
Its holiday
petrol-powered vehicle
20
```

4 Loops

- for (iterates overs sequence of values or elements)
- while (while loop executes the block of code repeatedly as long as the specified condition is true)

[ ]:
```python
# for loop over list
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)

# for loop using range
for i in range(1,10): # is equivalent to for i=0;i<10;i++
    print(i)

# while loop
count = 0
while count<5:
    print(count)
    count+=1
```

```
1
2
3
4
5
1
2
3
4
5
6
7
8
9
0
1
2
3
4
```

5 Exception Handling

It allows us to detect, handle, and recover from unexpected situations without crashing the program. Python provides a structured way to handle exceptions using - try - except - else - finally

[ ]:
```
def divide_numbers(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed")
    else:
        print("Division result:", result)
    finally:
        print("Cleanup: This code always runs, regardless of exceptions")
divide_numbers(1,0)
```

```
Error: Division by zero is not allowed
Cleanup: This code always runs, regardless of exceptions
```

6 Function

- In Python, functions are reusable blocks of code that perform a specific task.
- They provide a way to modularize code, improve readability, and promote code reuse.
- Functions can take input parameters (arguments), perform operations, and optionally return a result.

6.0.1 Define a Function

```
#Syntax
def function_name(parameters):
    # Block of code
    return expression
```

[ ]:
```
def greet(name):
    return f"Hello, {name}!"
```

6.0.2 Calling a function

Once defined, a function can be called or invoked to execute its code.

[ ]:
```
message = greet("Alice")
print(message)  # Output: Hello, Alice!
```

```
Hello, Alice!
```

6.0.3 Parameters and Arguments:

- Functions can accept zero or more parameters, which act as placeholders for values passed during function invocation.

- Parameters are specified in the function definition, while arguments are the actual values supplied when calling the function.

[ ]:
```
def add(x, y): # x and y are parameters
    return x + y

result = add(3, 5)  # x = 3, y = 5 are arguments supplied to the function
print(result)  # Output: 8
```

```
8
```

6.0.4 Default Arguments:

- We can specify default values for function parameters.
- If no value is provided for a default parameter, the default value is used.

[ ]:
```
def greet(name="World"):
    return f"Hello, {name}!"

greet()
```

[ ]:
```
'Hello, World!'
```

6.0.5 Arbitrary Arguments:

- Functions can accept a variable number of arguments using *args or **kwargs (keyword arguments).
- *args is used to pass a variable number of positional arguments, while **kwargs is used to pass a variable number of keyword arguments.

[ ]:
```
def add(*args):
    return sum(args)

sum = add(1,2,3,4,5,6)
print(sum)
```

```
21
```

6.0.6 Return Statement:

- Functions can return a value using the return statement.
- If no return statement is provided, the function returns None by default.

[ ]:
```
def multiply(x, y):
    return x * y

product = multiply(4, 6)
print(product)  # Output: 24
```

```
24
```

6.0.7 Lambda Functions

- Lambda functions, also known as anonymous functions, are small, single-expression functions defined using the lambda keyword.
- They are often used for short, simple operations.

[ ]:
```
square = lambda x: x ** 2
print(square(4))  # Output: 16
```

```
16
```