Python Programming

Practical Questions - Database Management

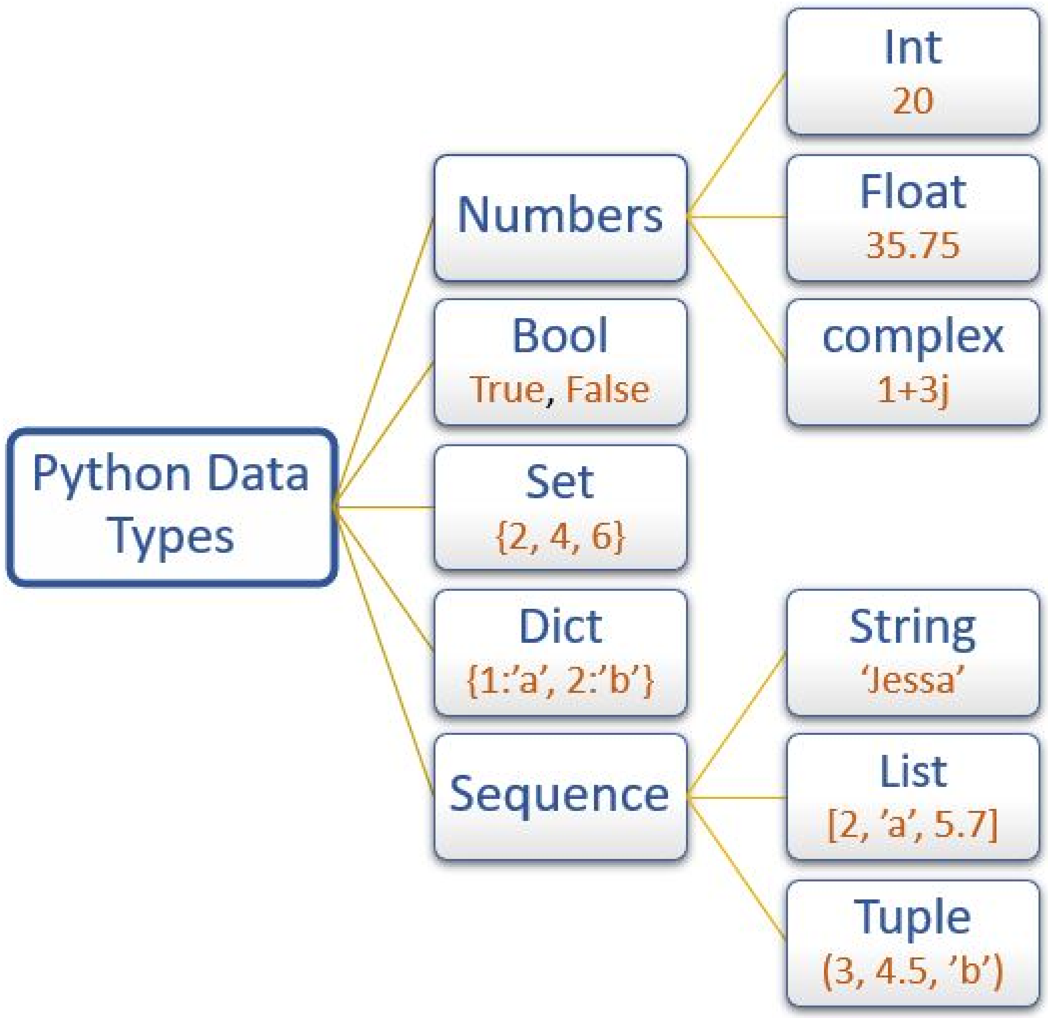
1. DATA-TYPES, 2. SETS, 3. COMMENTS, 4. KEYWORDS, 5. OPERATORS, 6. TYPE-CASTING

Data- Types in Python

Python stores numbers, strings, and a list of values using different data types.

There are mainly four types of basic/primitive data types available in Python

- **Numeric**: int, float, and complex
- **Sequence**: String, list, and tuple
- **Set**
- **Dictionary** (dict)



To check the data type of variable use the built-in function `type()` and `isinstance()`.

- The `type()` function returns the data type of the variable
- The isinstance() function checks whether an object belongs to a particular class.

Data type	Description	Example
`int`	To store integer values	`n = 20`
`float`	To store decimal values	`n = 20.75`
`complex`	To store complex numbers (real and imaginary part)	`n = 10+20j`
str	To store textual/string data	`name = 'Jessa'`
bool	To store boolean values	`flag = True`
list	To store a sequence of mutable data	`l = [3, 'a', 2.5]`
tuple	To store sequence immutable data	`t =(2, 'b', 6.4)`
dict	To store key: value pair	`d = {1:'J', 2:'E'}`
set	To store unorder and unindexed values	`s = {1, 3, 5}`
frozenset	To store immutable version of the set	`f_set=frozenset({5,7})`
range	To generate a sequence of number	`numbers = range(10)`
bytes	To store bytes values	`b=bytes([5,10,15,11])`

Str data type

In Python, A string is a **sequence of characters enclosed within a single quote or double quote**. These characters could be anything like letters, numbers, or special symbols enclosed within double quotation marks. For example, `"PYnative"` is a string.

The string type in Python is represented using a `str` class.

To work with text or character data in Python, we use Strings. Once a string is created, we can do many operations on it, such as searching inside it, creating a substring from it, and splitting it.

Example

```
platform = "PYnative"
print(type(platform))  # <class 'str'>

# display string
print(platform)  # 'PYnative'

# accessing 2nd character of a string
print(platform[1])  # Y
```

Note: The string is immutable, i.e., it can not be changed once defined. You need to create a copy of it if you want to modify it. This non-changeable behavior is called immutability.

Example

```
platform = "PYnative"
# Now let's try to change 2nd character of a string.
platform[0] = 'p'
# Gives TypeError: 'str' object does not support item assignment
```

Int data type

Python uses the int data type to **represent whole integer values**. For example, we can use the int data type to store the roll number of a student. The Integer type in Python is represented using a `int` class.

You can store positive and negative integer numbers of any length such as 235, -758, 235689741.

We can create an integer variable using the two ways

1. Directly assigning an integer value to a variable
2. Using a `int()` class.

Example

```
# store int value
roll_no = 33
# display roll no
print("Roll number is:", roll_no)
# output 33
print(type(roll_no))
# output class 'int'

# store integer using int() class
id = int(25)
print(id)  # 25
print(type(id))  # class 'int'
```

You can also store integer values other than base 10 such as

- Binary (base 2)
- Octal (base 8)
- Hexadecimal numbers (base 16)

Float data type

To represent **floating-point values or decimal value**s, we can use the float data type. For example, if we want to store the salary, we can use the float type.

The float type in Python is represented using a `float` class.

We can create a float variable using the two ways

1. Directly assigning a float value to a variable
2. Using a `float()` class.

Example

```
# store a floating-point value
salary = 8000.456
print("Salary is :", salary)  # 8000.456
print(type(salary))  # class 'float'

# store a floating-point value using float() class
num = float(54.75)
print(num)  # 54.75
print(type(num))  # class 'float'
```

Floating-point values can be represented using the exponential form, also called **scientific notation.** The benefit of using the exponential form to represent floating-point values is we can represent large values using less memory.

List Data Type

The Python List is an **ordered collection (also known as a sequence) of elements**. List elements can be accessed, iterated, and removed according to the order they inserted at the creation time.

We use the list data type to represent groups of the element as a single entity. For example: If we want to store all student's names, we can use `list` type.

1. The list can contain data of all data types such as `int`, `float`, `string`
2. Duplicates elements are allowed in the list
3. The list is mutable which means we can modify the value of list elements

We can create a list using the two ways

1. By enclosing elements in the **square brackets** `[]`.
2. Using a `list()` class.

Example list creation and manipulation

```
my_list = ["Jessa", "Kelly", 20, 35.75]
# display list
print(my_list)  # ['Jessa', 'Kelly', 20, 35.75]
print(type(my_list))  # class 'list'

# Accessing first element of list
print(my_list[0])  # 'Jessa'

# slicing list elements
print(my_list[1:5])  # ['Kelly', 20, 35.75]
```

```python
# modify 2nd element of a list
my_list[1] = "Emma"
print(my_list[1])  # 'Emma'

# create list using a list class
my_list2 = list(["Jessa", "Kelly", 20, 35.75])
print(my_list2)  # ['Jessa', 'Kelly', 20, 35.75]
```

Tuple data type

Tuples are **ordered collections of elements that are unchangeab**le. The `tuple` is the same as the `list`, except the tuple is immutable means we can't modify the tuple once created.

In other words, we can say a tuple is a read-only version of the list.

For example: If you want to store the roll numbers of students that you don't change, you can use the `tuple` data type.

Note: Tuple maintains the insertion order and also, allows us to store duplicate elements.

We can create a tuple using the two ways

1. By enclosing elements in the parenthesis ()
2. Using a `tuple()` class.

Example Tuple creation and manipulation

```python
# create a tuple
my_tuple = (11, 24, 56, 88, 78)
print(my_tuple)  # (11, 24, 56, 88, 78)
print(type(my_tuple))  # class 'tuple'

# Accessing 3rd element of a tuple
print(my_tuple[2])  # 56

# slice a tuple
print(my_tuple[2:7])  # (56, 88, 78)

# create a tuple using a tuple() class
my_tuple2 = tuple((10, 20, 30, 40))
print(my_tuple2)  # (10, 20, 30, 40)
```

Tuple is immutable

A tuple is immutable means once we create a tuple, we can't modify it

```python
# create a tuple
my_tuple = (11, 24, 56, 88, 78)

# modify 2nd element of tuple
my_tuple[1] = 35
print(my_tuple)
# TypeError: 'tuple' object does not support item assignment
```

Dict data type

In Python, dictionaries are **unordered collections of unique values stored in (Key-Value) pairs**. Use a dictionary data type to store data as a key-value pair.

The dictionary type is represented using a `dict` class. For example, If you want to store the name and roll number of all students, then you can use the `dict` type.

In a dictionary, duplicate keys are not allowed, but the value can be duplicated. If we try to insert a value with a duplicate key, the old value will be replaced with the new value.

Dictionary has some characteristics which are listed below:

1. A heterogeneous (i.e., `str`, `list`, `tuple`) elements are allowed for both key and value in a dictionary. But An object can be a key in a dictionary if it is hashable.
2. The dictionary is mutable which means we can modify its items
3. Dictionary is unordered so we can't perform indexing and slicing

We can create a dictionary using the two ways

1. By enclosing key and values in the curly brackets `{}`
2. Using a `dict()` class.

Example dictionary creation and manipulation

```python
# create a dictionary
my_dict = {1: "Smith", 2: "Emma", 3: "Jessa"}

# display dictionary
print(my_dict)  # {1: 'Smith', 2: 'Emma', 3: 'Jessa'}
print(type(my_dict))  # class 'dict'

# create a dictionary using a dit class
my_dict = dict({1: "Smith", 2: "Emma", 3: "Jessa"})

# display dictionary
print(my_dict)  # {1: 'Smith', 2: 'Emma', 3: 'Jessa'}
print(type(my_dict))  # class 'dict'

# access value using a key name
print(my_dict[1])  # Smith
```

```
# change the value of a key
my_dict[1] = "Kelly"
print(my_dict[1])  # Kelly
```

Set Data Type

In Python, a set is an **unordered collection of data items that are unique**. In other words, Python Set is a collection of elements (Or objects) that contains no duplicate elements.

In Python, the Set data type used to represent a group of unique elements as a single entity. For example, If we want to store student ID numbers, we can use the set data type.

The Set data type in Python is represented using a `set` class.

We can create a Set using the two ways

1. By enclosing values in the curly brackets `{}`
2. Using a `set()` class.

The set data type has the following characteristics.

1. It is mutable which means we can change set items
2. Duplicate elements are not allowed
3. Heterogeneous (values of all data types) elements are allowed
4. Insertion order of elements is not preserved, so we can’t perform indexing on a Set

Example Set creation and manipulation

```
# create a set using curly brackets{,}
my_set = {100, 25.75, "Jessa"}
print(my_set)  # {25.75, 100, 'Jessa'}
print(type(my_set))  # class 'set'

# create a set using set class
my_set = set({100, 25.75, "Jessa"})
print(my_set)  # {25.75, 100, 'Jessa'}
print(type(my_set))  # class 'set'

# add element to set
my_set.add(300)
print(my_set)  # {25.75, 100, 'Jessa', 300}

# remove element from set
my_set.remove(100)
print(my_set)  # {25.75, 'Jessa', 300}
```

Bool data type

In Python, to **represent boolean values (`True` and `False`)** we use the `bool` data type. Boolean values are used to evaluate the value of the expression. For example, when we compare two values, the expression is evaluated, and Python returns the boolean `True` or `False`.

Example

```
x = 25
y = 20

z = x > y
print(z)  # True
print(type(z))  # class 'bool'
```

Bytes data type

The `bytes` data type represents a group of byte numbers just like an array. We use the `bytes()` constructor to create bytes type, which also returns a bytes object. Bytes are **immutable** (Cannot be changed).

Use bytes data type if we want to handle binary data like images, videos, and audio files.

Example

```
a = [9, 14, 17, 11, 78]
b = bytes(a)
print(type(b))  # class 'bytes'
print(b[0])  # 9
print(b[-1])  # 78
```

In bytes, **allowed values are 0 to 256**. If we are trying to use any other values, then we will get a `ValueError`.

Example

```
a = [999, 314, 17, 11, 78]  # Gives error range must be in 0 to 256
b = bytes(a)
print(type(b))
# ValueError: bytes must be in range(0, 256)
```

Range data type

In Python, The built-in function `range()` used to generate a sequence of numbers from a start number up to the stop number. For example, If we want to represent the roll number from 1 to 20, we can use the `range()` type. By default, it returns an iterator object that we can iterate using a `for loop`.

Example

```python
# Generate integer numbers from 10 to 14
numbers = range(10, 15, 1)
print(type(numbers))  # class 'range'

# iterate range using for loop
for i in range(10, 15, 1):
    print(i, end=" ")
# Output 10 11 12 13 14
```

Python Comments

Comments describe what is happening inside a program so that a person looking at the source code does not have difficulty figuring it out.

In Python, we **use the hash (`#`) symbol to start writing a comment**.

Single-line Comment

Python has two types of comments single-line and multi-line comments.

In Python, single-line comments are indicated by a hash sign(`#`). The interpreter ignores anything written after the # sign, and it is effective till the end of the line.

```python
# welcome message
print('Welcome to PYnative...')
```

Multi-Line Comments

In Python, there is no separate way to write a multi-line comment. Instead, we need to use a hash sign at the beginning of each comment line to make it a multi-line comment

Example

```python
# This is a
# multiline
# comment
print('Welcome to PYnative...')
```

We can use the following two ways to get the list of keywords in Python

- **keyword module**: The keyword is the buil-in module to get the list of keywords. Also, this module allows a Python program to determine if a string is a keyword.
- `help()` **function**: Apart from a keyword module, we can use the `help()` function to get the list of keywords

```
import keyword
print(keyword.kwlist)
```

All the keywords except, `True`, `False`, and `None`, must be written in a lowercase alphabet symbol.

Example 2: The `help()` function

```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               break               for                 not
None                class               from                or
True                continue            global              pass
__peg_parser__      def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
```

Operators

Arithmetic operator

Arithmetic operators are the most commonly used. The Python programming language provides arithmetic operators that perform addition, subtraction, multiplication, and division. It works the same as basic mathematics.

There are seven arithmetic operators we can use to perform different mathematical operations, such as:

1. `+` (Addition)
2. `-` (Subtraction)
3. `*` (Multiplication)
4. `/` (Division)
5. `//` Floor division)
6. `%` (Modulus)
7. `**` (Exponentiation)

Relational (comparison) operators

Relational operators are also called comparison operators. It performs a comparison between two values. It returns a boolean True or False depending upon the result of the comparison.

Python has the following six relational operators.

Assume variable `x` holds 10 and variable `y` holds 5

Operator	Description	Example
`>` (Greater than)	It returns True if the left operand is greater than the right	`x > y` result is `True`
`<` (Less than)	It returns True if the left operand is less than the right	`x < y` result is `False`
`==` (Equal to)	It returns True if both operands are equal	`x == y` result is `False`
`!=` (Not equal to)	It returns True if both operands are equal	`x != y` result is `True`
`>=` (Greater than or equal to)	It returns True if the left operand is greater than or equal to the right	`x >= y` result is `True`
`<=` (Less than or equal to)	It returns True if the left operand is less than or equal to the right	`x <= y` result is `False`

Assignment operators

In Python, Assignment operators are used to assigning value to the variable. Assign operator is denoted by = symbol. For example, `name = "Jessa"` here, we have assigned the string literal 'Jessa' to a variable name.

Also, there are shorthand assignment operators in Python. For example, `a+=2` which is equivalent to `a = a+2`.

Operator	Meaning	Equivalent
`=` (Assign)	`a=5` Assign 5 to variable `a`	a = 5
`+=` (Add and assign)	`a+=5` Add 5 to a and assign it as a new value to `a`	a = a+5
`-=` (Subtract and assign)	`a-=5` Subtract 5 from variable `a` and assign it as a new value to `a`	a = a-5
`*=` (Multiply and assign)	`a*=5` Multiply variable `a` by 5 and assign it as a new value to `a`	a = a*5
`/=` (Divide and assign)	`a/=5` Divide variable `a` by 5 and assign a new value to `a`	a = a/5

Operator	Meaning	Equivalent
`%=` (Modulus and assign)	`a%=5` Performs modulus on two values and assigns it as a new value to `a`	a = a%5
`**=` (Exponentiation and assign)	`a**=5` Multiply `a` five times and assigns the result to `a`	a = a**5
`//=` (Floor-divide and assign)	`a//=5` Floor-divide `a` by 5 and assigns the result to `a`	a = a//5

Logical operators

Logical operators are useful when checking a condition is `true` or not. Python has three logical operators. All logical operator returns a boolean value `True` or `False`depending on the condition in which it is used.

Operator	Description	Example
`and` (Logical and)	True if both the operands are True	a and b
`or` (Logical or)	True if either of the operands is True	a or b
`not` (Logical not)	True if the operand is False	not a

Identity operators

Use the Identity operator to check whether the value of two variables is the same or not. This operator is known as a **reference-quality operator** because the identity operator compares values according to two variables' memory addresses.

Python has 2 identity operators `is` and `is not`.

`is` operator

The `is` operator returns Boolean `True` or `False`. It Return `True` if the memory address first value is equal to the second value. Otherwise, it returns `False`.

Bitwise Operators

In Python, bitwise operators are used to performing bitwise operations on integers. To perform bitwise, we first need to convert integer value to binary (0 and 1) value.

The bitwise operator operates on values bit by bit, so it's called **bitwise**. It always returns the result in decimal format. Python has 6 bitwise operators listed below.

1. `&` Bitwise and
2. `|` Bitwise or
3. `^` Bitwise xor

4. `~` Bitwise 1's complement
5. `<<` Bitwise left-shift
6. `>>` Bitwise right-shift

Python Casting: Type Conversion and Type Casting

In Python, we can convert one type of variable to another type. This conversion is called type casting or type conversion.

In casting, we convert variables declared in specific data types to the different data types.

Python performs the following two types of casting.

- **Implicit casting**: The Python interpreter automatically performs an implicit Type conversion, which avoids loss of data.
- **Explicit casting**: The explicit type conversion is performed by the user using built-in functions.

To perform a type casting, we are going to use the following built-in functions

1. `int()`**:** convert any type variable to the integer type.
2. `float()`**:** convert any type variable to the float type.
3. `complex()`**:** convert any type variable to the complex type.
4. `bool()`**:** convert any type variable to the bool type.
5. `str()`**:** convert any type variable to the string type.

In type casting, data loss may occur because we enforce the object to a specific data type.

Int type conversion

In `int` type conversion, we use the `int()` function to convert variables of other types to `int` type. Variable can be of any type such as `float`, `string`, `bool`.

While performing `int` type conversion, we need to remember the following points.

1. When converting **string type** to **int type**, a string must contain integral value only and should be base-10.
2. We can convert any type to `int` type, but we cannot perform **complex** to int type.

Casting float value to an integer

```python
pi = 3.14  # float number
print(type(pi))
# Output class 'float'

# converting float integer
num = int(pi)
print("Integer number:", num)
# Output  3
print(type(num))
# Output class 'int'
```

Casting Boolean value to an integer

```python
flag_true = True
flag_false = False
print(type(flag_true))
# Output class 'bool'

# converting boolean to integer
num1 = int(flag_true)
num2 = int(flag_false)

print("Integer number 1:", num1)
# Output 1
print(type(num1))
# Output class 'int'

print("Integer number 2:", num2)
# 0
print(type(num2))
# class 'int'
```

Casting a string to an integer

```python
string_num = "225"
print(type(string_num))
# Output class 'str'

# converting str to integer
num1 = int(string_num)

print("Integer number 1:", num1)
# Output 225
print(type(num1))
# Output class 'int'
```

When converting **string type** to **int type**, a string must contain integral value only and should be base-10. If you try to convert

Example

```python
string_num = 'Score is 25'
print(type(string_num))
# Output class 'str'

# ValueError: invalid literal for int() with base 10: 'Score is 25'
num = int(string_num)
print(num)
```

Float type conversion

In float type conversion we use a built-in function `float()` . This function converts variables of other types to `float` types.

Casting integer to float

```python
num = 725
print(type(num))
# Output class 'int'

# converting float to integer
num1 = float(num)

print("Float number:", num1)
# Output 725.0
print(type(num1))
# Output class 'float'
```

**Casting* Boolean to float*

```python
flag_true = True
flag_false = False
print(type(flag_true))
# Output class 'bool'

# converting boolean to float
num1 = float(flag_true)
num2 = float(flag_false)

print("Float number 1:", num1)
# Output 1.0
print(type(num1))
# class 'float'

print("Float number 2:", num2)
# Output 0.0
print(type(num2))
# class 'float'
```

**Casting* string to float*

```
string_num = "725.535"
print(type(string_num))
# Output class 'str'

# converting str to float
num1 = float(string_num)

print("Float number:", num1)
# Output 725.535
print(type(num1))
# class 'float'
```

While performing float type conversion, we need to remember some points.

1. We can convert any type to `float` type, but we cannot cast **complex** to float type.
2. While converting **string type** to **float type**, a string must contain an **integer/decimal value** of base-10.

Casting integer to Boolean type

```
num1 = 10
num2 = 0
print(type(num1))  # class 'int'

# Convert into to bool
b1 = bool(num1)
b2 = bool(num2)

print(b1)
# Output True
print(b2)
# Output False

print(type(b1))
# class 'bool'
```

Casting float to Boolean type

```
f_num1 = 25.35
f_num2 = 0.0
print(type(f_num1))  # class 'float'

# Convert float into to bool
b1 = bool(f_num1)
b2 = bool(f_num2)

print(b1)
# Output True

print(b2)
```

```
# Output False

print(type(b1))
# Output class 'bool'
```

Casting string to Boolean type

```
s1 = "False"
s2 = "True"
s3 = "812"
s4 = ""
print(type(s1))  # class 'str'

# Convert string into to bool
b1 = bool(s1)
b2 = bool(s2)
b3 = bool(s3)
b4 = bool(s4)

print(b1)  # True
print(b2)  # True
print(b3)  # True
print(b4)  # False
print(type(b1))  # class 'bool'
```

String type conversion

In `str` type conversion, we use the built-in function `str()` to convert converts variables of other types to a string type. This function returns the string type of object (value).

Casting int to str type

```
num = 15
print(type(num))  # class 'int'

# converting int to str type
s1 = str(num)
print(s1)
# Output '15'
print(type(s1))
# Output class 'str'
```

Casting float type to str type

```
num = 75.35
print(type(num))  # class 'float'

# converting float to str type
s1 = str(num)
print(s1)
# Output '75.35'
print(type(s1))
# Output class 'str'
```

Casting bool type to str type

```
b1 = True
b2 = False
print(type(b1))  # class 'bool'

# converting bool to str type
s1 = str(b1)
s2 = str(b2)
print(s1)
# Output 'True'
print(s2)
# Output 'False'
print(type(s1))  # class 'str'
```