MODULE-1

PPT-3

Python Data Types

- Built-in Data Types

Text Type: str

Numeric Types: int, float, complex

Sequence Types: list, tuple, range

Mapping Type: dict

Set Types: set, frozenset

Boolean Type: bool

Binary Types: bytes, bytearray, memoryview

You can get the data type of any object by using the type() function:

```
x = 5

print(type(x)) # Print the data type of the variable x
```

Setting the Data Type

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

Setting the Specific Data Type

Example	Data Type
x = str("Hello World")	str
x = int(20)	int
x = float(20.5)	float
x = complex(1j)	complex
x = list(("apple", "banana", "cherry"))	list
x = tuple(("apple", "banana", "cherry"))	tuple
x = range(6)	range
x = dict(name="John", age=36)	dict
x = set(("apple", "banana", "cherry"))	set
x = frozenset(("apple", "banana", "cherry"))	frozenset
x = bool(5)	bool
x = bytes(5)	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

Python Numbers

- There are three numeric types in Python:

1. int
2. float
3. complex

- Variables of numeric types are created when you assign a value to them:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

Int- Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
x = 1
y = 35656222554887711
z = -3255522
```

Float

- Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

```
x = 1.10
y = 1.0
z = -35.59
```

- Float can also be scientific numbers with an "e" to indicate the power of 10.

```
x = 35e3
y = 12E4
z = -87.7e100
```

Complex

- Complex numbers are written with a "j" as the imaginary part:

```
x = 3+5j
y = 5j
z = -5j
```

Type Conversion

- You can convert from one type to another with the int(), float(), and complex() methods:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
#convert from int to float:
a = float(x)
#convert from float to int:
b = int(y)
#convert from int to complex:
c = complex(x)
print(a)
print(b)
print(c)
print(type(a))
print(type(b))
print(type(c))
```

```
>>> x = 1    # int
>>> y = 2.8  # float
>>> z = 1j   # complex
>>>
>>> #convert from int to float:
>>> a = float(x)
>>>
>>> #convert from float to int:
>>> b = int(y)
>>>
>>> #convert from int to complex:
>>> c = complex(x)
>>>
>>> print(a)
1.0
>>> print(b)
2
>>> print(c)
(1+0j)
>>>
>>> print(type(a))
<class 'float'>
>>> print(type(b))
<class 'int'>
>>> print(type(c))
<class 'complex'>
```

Random Number

- Python does not have a random() function to make a random number, but Python has a built-in module called random that can be used to make random numbers:
- Import the random module, and display a random number between 1 and 9:

```
import random
print(random.randrange(1, 10))
```

Python Casting

- There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.
- Casting in python is therefore done using constructor functions:

1. int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
2. float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
3. str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Example

Integers:

```
x = int(1)
y = int(2.8)
z = int("3")
print(x)
print(y)
print(z)
```

Floats-

```
x = float(1)
y = float(2.8)
z = float("3")
w = float("4.2")
print(x)
print(y)
print(z)
print(w)
```

Strings:

```
x = str("s1")
y = str(2)
z = str(3.0)
print(x)
print(y)
print(z)
```

Using Python as a Calculator- Numbers

- The interpreter acts as a simple calculator: you can type an expression at it and it will write the value.
- Expression syntax is straightforward: the operators +, - , * and / work just like in most other languages, parentheses (()) can be used for grouping. For example:

```
>>> 2+2
4
>>> 50-5*6
20
>>> (50-5*6)/4
5.0
>>> 8/5 # division always returns a floating point number
1.6
```

Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

Example

```
>>> 17 / 3 # classic division returns a float 5.666666666666667
>>> 17 // 3 # floor division discards the fractional part
5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # result * divisor + remainder 17
```

- With Python, it is possible to use the ** operator to calculate powers 1:

```
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

- The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

- There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 4 * 3.75 - 1 14.0
```

- In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax 12.5625
>>> price + _ 113.0625
>>> round(_, 2)
13.06
```

- This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.
- In addition to int and float, Python supports other types of numbers, such as Decimal and Fraction. Python also has built-in support for complex numbers, and uses the j or J suffix to indicate the imaginary part (e.g. 3+5j).

Strings

- Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result. \ can be used to escape quotes:
- Examples-

'spam eggs' *# single quotes*

'doesn\'t' *# use \' to escape the single quote...*

"doesn't" *# ...or use double quotes instead*

'"Yes," they said.'

"\"Yes,\" they said.“

'"Isn\'t," they said.'

String Literals

- String literals in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

- You can display a string literal with the print() function:

```
print("Hello")
print('Hello')
```

- Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

```
a = "Hello"
print(a)
```

- You can assign a multiline string to a variable by using three quotes: You can use three double quotes or three single quotes:

```
a = """Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua."""
print(a)
```

- If you don’t want characters prefaced by \ to be interpreted as special characters, you can use *raw strings* by adding an r before the first quote:

```
print('C:\some\name') # here \n means newline!
print(r'C:\some\name') # note the r before the quote
```

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

- Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
# 3 times 'hi', followed by 'hello'
3 * 'hi' + 'hello'
```

```
>>> # 3 times 'hi', followed by 'hello'
>>> 3 * 'hi' + 'hello'
'hihihihello'
```

- Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
'Py' 'thon‘
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
word = 'Python'
word[0] # character in position 0
word[5] # character in position 5
```

Indices may also be negative numbers, to start counting from the right:

```
word=‘python’
word[-1] # last character
word[-2] # second-last character
word[-6]
```

Note that since -0 is the same as 0, negative indices start from -1

- In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain substring:

```
word[0:2] # characters from position 0 (included) to 2 (excluded)
word[2:5] # characters from position 2 (included) to 5 (excluded)
```

- Note how the start is always included, and the end always excluded. This makes sure that s[:i] + s[i:] is always equal to s:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

- Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2] # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:] # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

- Attempting to use an index that is too large will result in an error:

word[42] *# the word only has 6 characters*

However, out of range slice indexes are handled gracefully when used for slicing:

>>> word[4:42]

'on'

>>> word[42:]

''

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

word[0] = 'J'

word[2:] = 'py'

- If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython' >>>
word[:2] + 'py'
'Pypy'
```

- The built-in function len() returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
```

-