Python Fundamentals day 5

Today's Agenda

- Typecasting
- Mutable and Immutable datatypes
- Overview of datatypes

Typecasting

The process of converting one type of data to another type in programming language is called as typecasting. In python typecasting is done using the functions already present in python these functions are called as built-in functions.

int( ) **str( )** **list( )** **set( )** **hex( )** **chr( )**

float( ) **complex( )** **tuple( )** **dict( )** **oct( )** **ord( )**

From the above mentioned built-in functions let's focus on the basic datatypes for now, rest of them we shall see in later on sessions.

- **First conversion:**

Integer ⟶ **Float**

float( )

Integer value must be converted to a floating point value, if this must happen then we have to use a function which converts integer type data to float type data. Which is none other than float( ).

```
In [1]: a=99

In [2]: print(a)
99

In [3]: print(type(a))
<class 'int'>

In [4]: b=float(a)

In [5]: print(b)
99.0

In [6]: print(type(b))
<class 'float'>
```

- **Second conversion:**

Float ⟶ Integer

int( )

Let's reverse the order now by converting floating type data to integer type data by using int( ).

```
In [7]: a=99.9

In [8]: print(a)
99.9

In [9]: print(type(a))
<class 'float'>

In [10]: b=int(a)

In [11]: print(b)
99

In [12]: print(type(b))
<class 'int'>
```

Note: We observe that 0.9 is lost during the conversion, because integer type data will not store decimal point values. Therefore we have to be very careful during type casting. In some cases 0.9 is a very huge value whereas in some other cases it is completely fine.

- **Third conversion:**

Float ⟶ Complex

complex( )

Let us convert a floating number to a complex number which contains a real number and an imaginary number.

```
In [13]: a=99.9

In [14]: print(a)
99.9

In [15]: print(type(a))
<class 'float'>

In [16]: b=complex(a)

In [17]: print(b)
(99.9+0j)

In [18]: print(type(b))
<class 'complex'>
```

We can also give an imaginary number as shown below

```
In [19]: c=complex(a,4)

In [20]: print(c)
(99.9+4j)
```

- **Fourth conversion:**

Integer ⟶ String

str( )

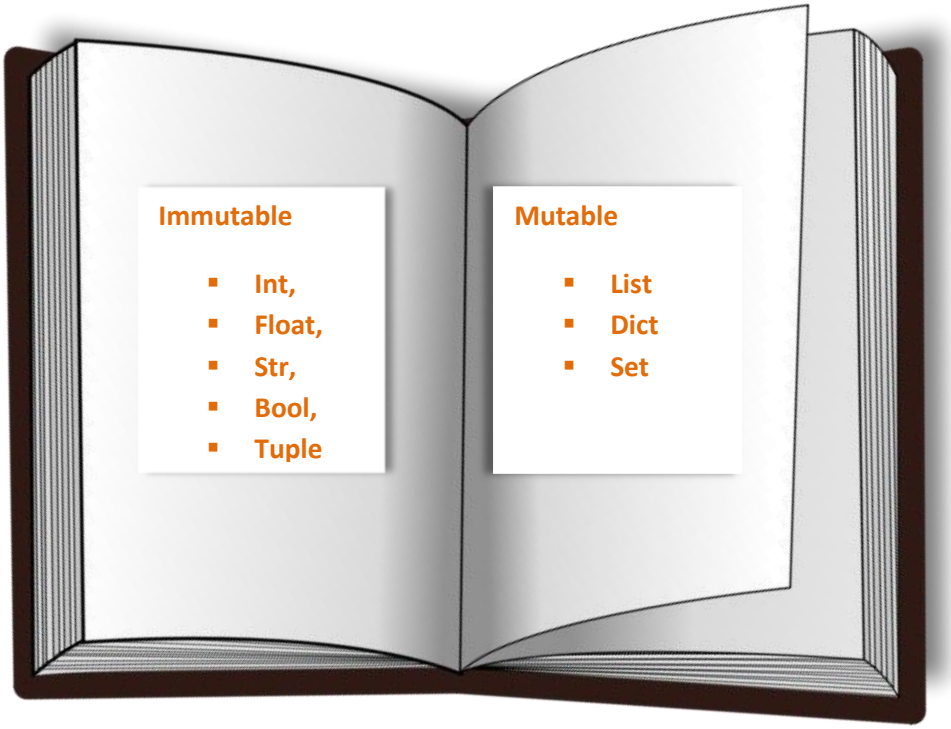
We have many other conversions which we will perform as and when needed. But let us see the last conversion which is most commonly used, integer type data to string type.

```
In [21]: a=99

In [22]: print(a)
99

In [23]: print(type(a))
<class 'int'>

In [24]: b=str(a)

In [25]: print(b)
99

In [26]: print(type(b))
<class 'str'>
```

String type data can certainly store numbers as well. We can know the type by printing it.

Mutable & Immutable datatypes

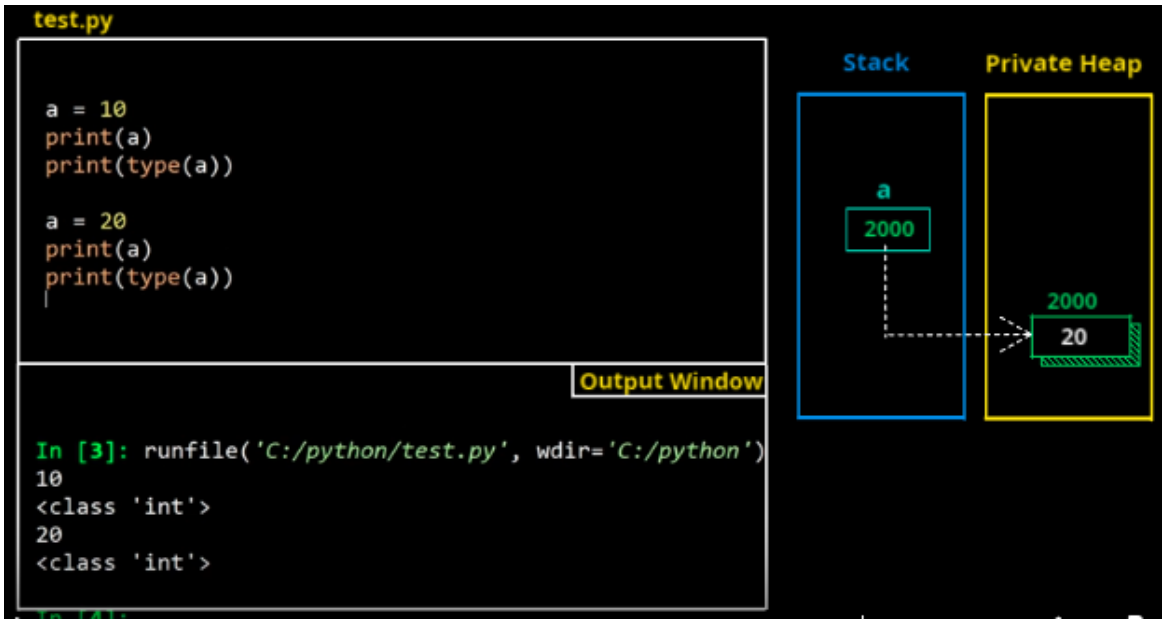
As we already know that mutable means changeable and immutable means non changeable. The datatypes whose values cannot be changed are called immutable datatypes and likewise the datatypes whose values can be changed are called mutable datatypes.



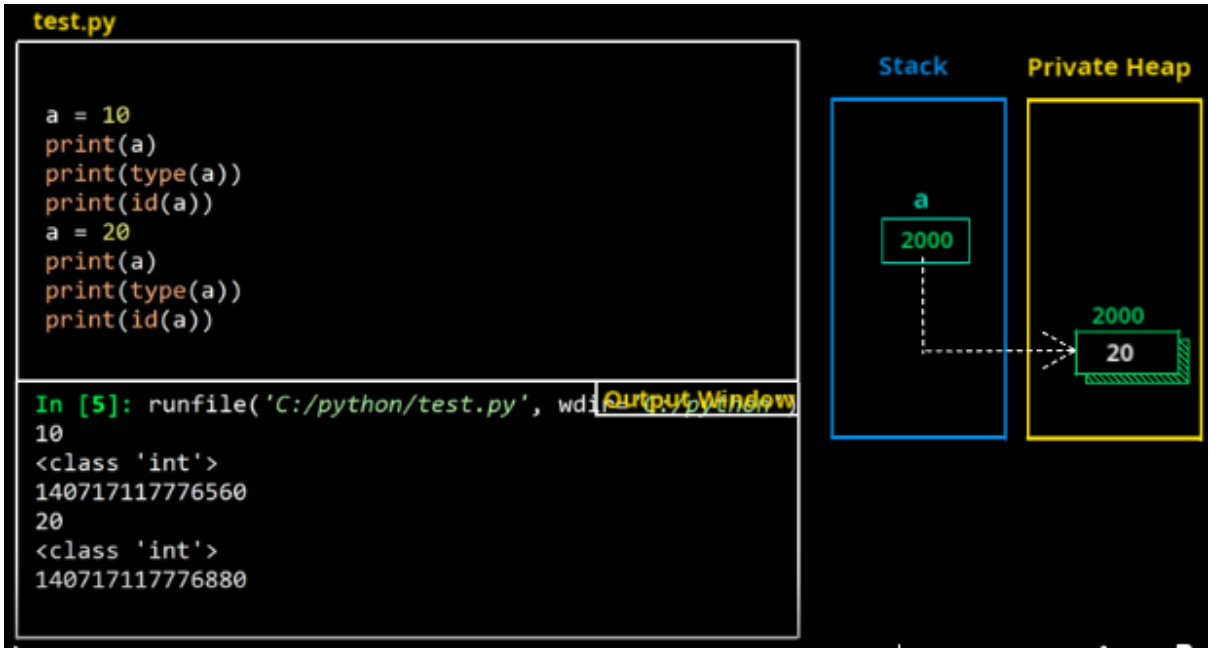
- **Immutable datatypes – (Integer)**

  Let us try assigning a value to variable a and try to change values.

  In below diagram we can see that a value gets changed but internally two different objects are created with different address. So the variable a is now pointing to the new value.

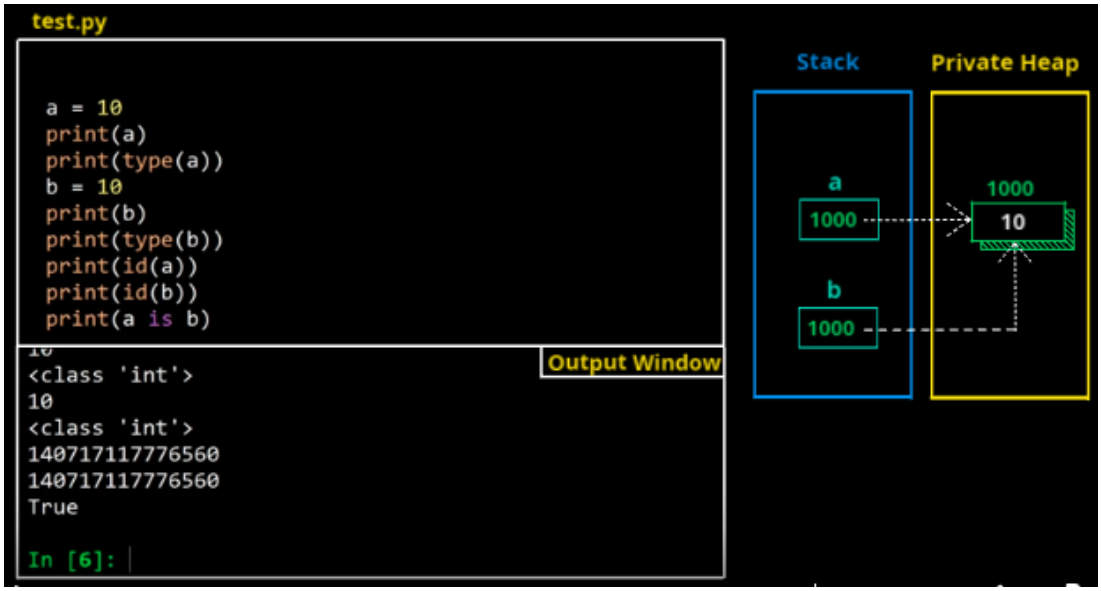


Let us try to verify the above statement by printing the address of both the objects.



We can see from the above diagram that both objects have different references. Which implies, when we try to modify the values assigned to a variable, new object gets created instead of modifying the existing value.

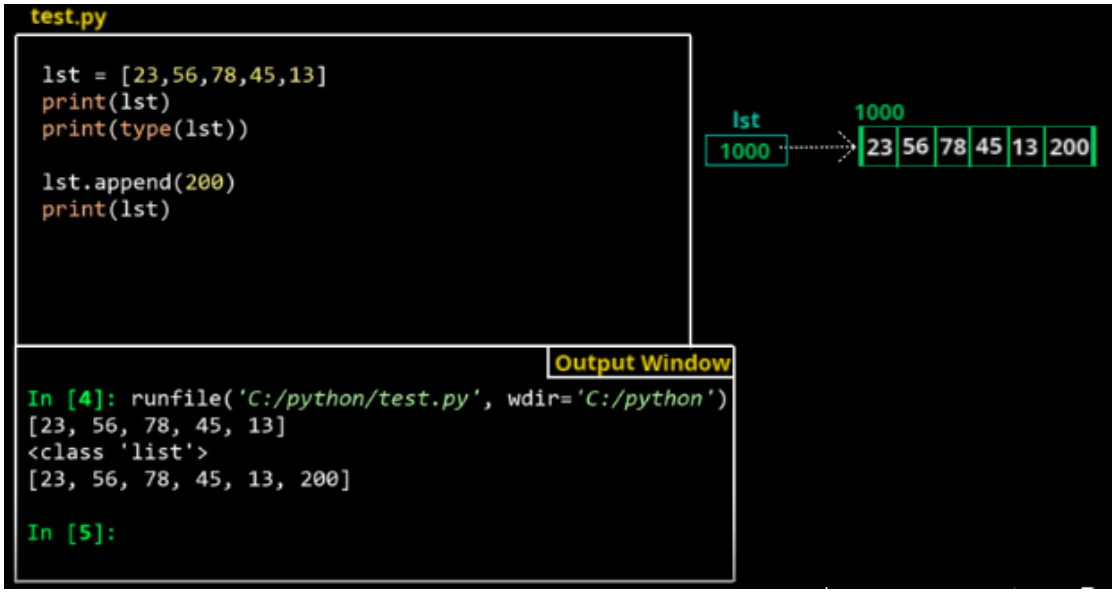
It is the same feature of immutability of certain datatypes which makes python memory efficient in the background. Which simply means that, when two variables are assigned the same value, two objects does not get created on private heap instead both the variables will be pointing to the same object. Because by now we know that some datatypes are immutable and their values cannot be changed. This is demonstrated in the below diagram,



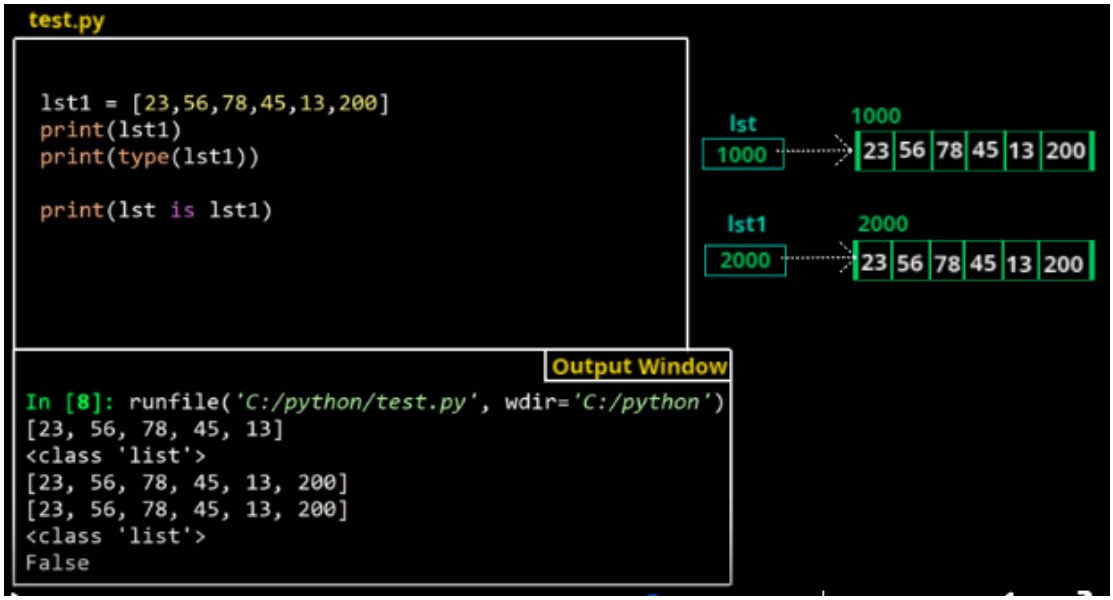
Above we have shown with respect to integer type data, whereas the same works for float, bool, string and tuple also because all these belong to immutable type data. And certainly we have also checked for equality of two variables by is operator.

- **Mutable datatype – (list)**

Let us create a list having some set of values and try to append new values. If you have gone through previous sessions you know that certainly that is possible in lists. Let's see it either ways



Let us now try to create another list with same values and see if memory is efficient in this case as well



We can see that when new list lst1 was equated to lst it resulted as false (Equality basically checks the references). This is because lists are mutable. So if any changes are made to one list the other list might get affected too. That's the reason no two lists type data will be pointing to a same lists object.

Overview of datatypes