Unit 1 -Introduction to Python and Jupyter Notebooks

Basic Data Types

int, float, str, bool

In [ ]:
```
a = 7
type(a)
```

Out[ ]:
```
int
```

In [ ]:
```
b = 7.3
type(b)
```

Out[ ]:
```
float
```

In [ ]:
```
c = 'abc'
type(c)
```

Out[ ]:
```
str
```

In [ ]:
```
d = True
type(d)
```

Out[ ]:
```
bool
```

Typecasting

Typecasting int

In [ ]:
```
a = 7
b = float(a)
print(b)
print(type(b))
```

```
7.0
<class 'float'>
```

In [ ]:
```
a = 7
b = str(a)
print(b)
print(type(b))
```

```
7
<class 'str'>
```

For typecasting int to bool, any number other than 0 is changed to True

In [ ]:
```
a = 7
b = bool(a)
print(b)
print(type(b))
```

```
True
<class 'bool'>
```

In [ ]:
```
a = 0
b = bool(a)
print(b)
print(type(b))
```

```
False
<class 'bool'>
```

Typecasting float

In [ ]:
```
a = 4.3
b = int(a)
print(b)
print(type(b))
```

```
4
<class 'int'>
```

In [ ]:
```
a = 4.3
b = str(a)
print(b)
print(type(b))
```

```
4.3
<class 'str'>
```

Typecasting float to bool has the same behaviour as int to bool. Everything other than 0.0 gets converted to True

In [ ]:
```
a = 4.3
b = bool(a)
print(b)
print(type(b))
```

```
True
<class 'bool'>
```

In [ ]:
```
a = 0.0
b = bool(a)
print(b)
print(type(b))
```

```
False
<class 'bool'>
```

Typecasting str

str can be typecast to int or float if it contains only numbers.

In [ ]:
```
a = 'abc123'
b = int(a)
print(b)
print(type(b))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-1e74953c6eb2> in <module>
      1 a = 'abc123'
----> 2 b = int(a)
      3 print(b)
      4 print(type(b))

ValueError: invalid literal for int() with base 10: 'abc123'
```

In [ ]:
```
a = '123'
b = int(a)
print(b)
print(type(b))
```

```
123
<class 'int'>
```

In [ ]:
```
a = '123.43'
b = float(a)
print(b)
print(type(b))
```

```
123.43
<class 'float'>
```

While typecasting str to bool, it always gets converted to True unless the string is empty

In [ ]:
```
a = 'abc123'
b = bool(a)
print(b)
print(type(b))
```

```
True
<class 'bool'>
```

In [ ]:
```
a = ''
b = bool(a)
print(b)
print(type(b))
```

```
False
<class 'bool'>
```

Typecasting bool

In [ ]:
```
a = True
b = int(a)
print(b)
print(type(b))
```

```
1
<class 'int'>
```

In [ ]:
```
a = True
b = float(a)
print(b)
print(type(b))
```

```
1.0
<class 'float'>
```

In [ ]:
```
a = True
b = str(a)
print(b)
print(type(b))
```

```
True
<class 'str'>
```

Comments

Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code.

Single line comments start with a #

Multi-line comments can be enclosed within triple double/single quotes.

In [ ]:
```
#This is a single line comment
"""This is
a multiple lines comment"""
```

Out[ ]:
```
'This is \na multiple lines comment'
```

Variables

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Rules for Python variables:

A variable name must start with a letter or the underscore character

A variable name cannot start with a number

A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)

Variable names are case-sensitive (age, Age and AGE are three different variables)

In [ ]:
```
#Legal variable names:
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

In [ ]:
```
#Illegal variable names:
2myvar = "John"
my-var = "John"
my var = "John"
```

```
  File "<ipython-input-39-2e3489f6a2b6>", line 2
    2myvar = "John"
         ^
SyntaxError: invalid syntax
```

Following is a list of keywords that should not be used as variable names in Python

In [ ]:
```
import keyword
keyword.kwlist
```

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

Input Output

In Python input from the user is taken using the input function... It accepts only one parameter called prompt which is a string message and always returns a strings

In [ ]:
```
x = input('Enter your input: ')
print(x)
```

```
Enter your input: Hello World!
Hello World!
```

The python print statement accepts the following paramters

Parameter	Description
object(s)	Any object, and as many as you like. Will be converted to string before printed
sep='*separator*'	Optional. Specify how to separate the objects, if there is more than one. Default is ' '
end='*end*'	Optional. Specify what to print at the end. Default is '\n' (line feed)
file	Optional. An object with a write method. Default is sys.stdout
flush	Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False

Note: The file parameter will be introduced in one of the later units. The flush parameter is beyond the scope of this syllabus.

In [ ]:
```
a = 1
b = 2
c = 3
print(a,b,c)
```

```
1 2 3
```

In [ ]:
```
print('hello')
```

```
hello
```

sep can be used to decide how the output is separated...

In [ ]:
```
print(*'hello', sep='-')
```

```
h-e-l-l-o
```

In [ ]:
```
a = 1
b = 2
c = 3
print(a,b,c, sep='*')
```

```
1*2*3
```

end can be used to specify how the output will end... the default is \n

In [ ]:
```
#without end
print('hello')
print('world')
```

```
hello
world
```

In [ ]:
```
#end with tab
print('hello', end='\t')
print('world')
```

```
hello	world
```

In [ ]:
```
#end with space
print('hello', end=' ')
print('world')
```

```
hello world
```

Operators

Arithmetic Operators

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation
//	Floor division

In [ ]:
```
a = 3
b = 4
print('Addition', a + b)
print('Subtraction', a - b)
print('Multiplication', a * b)
print('Division', a / b)
print('Modulus', a % b)
print('Exponentiation', a ** b)
print('Floor Division', a // b)
```

```
Addition 7
Subtraction -1
Multiplication 12
Division 0.75
Modulus 3
Exponentiation 81
Floor Division 0
```

Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3

In [ ]:
```
a = 4
a += 2
print(a)
```

```
6
```

In [ ]:
```
a -= 2
print(a)
```

```
4
```

In [ ]:
```
a *= 2
print(a)
```

```
8
```

In [ ]:
```
# Division operator always returns a float
a /= 2
print(a)
```

```
4.0
```

In [ ]:
```
a %= 2
print(a)
```

```
0.0
```

In [ ]:
```
a = 10
a **= 2
print(a)
```

```
100
```

In [ ]:
```
a //= 17
print(a)
```

```
5
```

Comparison Operators (always return a boolean value)

==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

In [ ]:
```
a = 4
b = 3
a == b
```

Out[ ]: False

In [ ]:
```
a < b
```

Out[ ]: False

In [ ]:
```
a > b
```

Out[ ]: True

In [ ]:
```
a <= b
```

Out[ ]: False

In [ ]:
```
a >= b
```

Out[ ]: True

In [ ]:
```
a != b
```

Out[ ]: True

Logical Operators

Operator	Description
and	Returns True if both statements are true
or	Returns True if one of the statements is true
not	Reverse the result, returns False if the result is true

In [ ]:
```
x = 0
y = 1
x==0 and y==0
```

Out[ ]: False

In [ ]:
```
x==0 or y==0
```

Out[ ]:
```
True
```

In [ ]:
```
not(y)
```

Out[ ]:
```
False
```

Membership Operators

Operator	Description
in	Returns True if a sequence with the specified value is present in the object
not in	Returns True if a sequence with the specified value is not present in the object

In [ ]:
```
x = 'hello'
'i' in x
```

Out[ ]:
```
False
```

In [ ]:
```
'i' not in x
```

Out[ ]:
```
True
```

Ternary Operators

```
[on_true] if [expression] else [on_false]
```

In [ ]:
```
a, b = 10, 20

# Copy value of a in min if a < b else copy b
min = a if a < b else b

print(min)
```

```
10
```

Nested Ternary Operator

In [ ]:
```
x = 0
y = 1
'1' if x else '2' if y else '3'
```

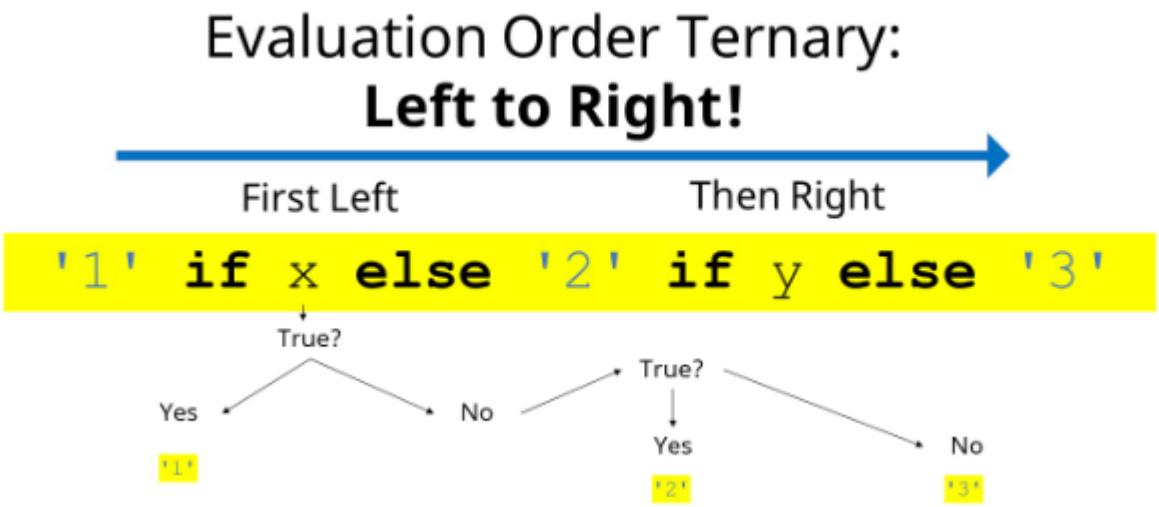
Out[ ]:
```
'2'
```

In [ ]:
```
x = 1
y = 0
'1' if x else '2' if y else '3'
```

Out[ ]: '1'

In [ ]:
```
x = 0
y = 0
'1' if x else '2' if y else '3'
```

Out[ ]: '3'



Operator Precedence

The following table illustrates the precedence order of the operators we have seen so far. It is in descending order (upper group has higher precedence than the lower ones).

Operators	Meaning
`()`	Parentheses
`**`	Exponent
`*`, `/`, `//`, `%`	Multiplication, Division, Floor division, Modulus
`+`, `-`	Addition, Subtraction
`==`, `!=`, `>`, `>=`, `<`, `<=`, `is`, `is not`, `in`, `not in`	Comparisons, Identity, Membership operators
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR

multiplication has higher precedence than subtraction

```
In [ ]: 10 - 4 * 2
```

```
Out[ ]: 2
```

But we can change this order using parentheses () as it has higher precedence than multiplication.

```
In [ ]: (10 - 4) * 2
```

```
Out[ ]: 12
```

Suppose we're constructing an if...else block which runs if when lunch is either fruit or sandwich and only if money is more than or equal to 2.

In [ ]:
```
# Precedence of or & and
meal = "fruit"

money = 0

if meal == "fruit" or meal == "sandwich" and money >= 2:
    print("Lunch being delivered")
else:
    print("Can't deliver lunch")
```

```
Lunch being delivered
```

This program runs if block even when money is 0. It does not give us the desired output since the precedence of and is higher than or.

We can get the desired output by using parenthesis () in the following way:

In [ ]:
```
# Precedence of or & and
meal = "fruit"

money = 0

if (meal == "fruit" or meal == "sandwich") and money >= 2:
    print("Lunch being delivered")
else:
    print("Can't deliver lunch")
```

```
Can't deliver lunch
```

Associativity of Python Operators

We can see in the above table that more than one operator exists in the same group. These operators have the same precedence.

When two operators have the same precedence, associativity helps to determine the order of operations.

Associativity is the order in which an expression is evaluated that has multiple operators of the same precedence. Almost all the operators have left-to-right associativity.

For example, multiplication and floor division have the same precedence. Hence, if both of them are present in an expression, the left one is evaluated first.

In [ ]:
```
# Left-right associativity
# Output: 3
print(5 * 2 // 3)

# Shows left-right associativity
# Output: 0
print(5 * (2 // 3))
```

```
3
0
```

Note: Exponent operator ** has right-to-left associativity in Python.

In [ ]:
```
# Shows the right-left associativity of **
# Output: 512, Since 2**(3**2) = 2**9
print(2 ** 3 ** 2)

# If 2 needs to be exponated fisrt, need to use ()
# Output: 64
print((2 ** 3) ** 2)
```

```
512
64
```

We can see that 2 **3** 2 is equivalent to 2 **(3** 2).