Lecture 02 - Data Types and Structures

Overview

In Python, **types** and **structures** are fundamental concepts that allow the storage, manipulation, and organization of data.

This notebook covers:

- **Basic data types**: `int`, `float`, `bool`, `str`
- **Data structures**: `tuple`, `list`, `set`, `dict`
- **Operations** and **built-in methods**

1. Basic Types

List of types

Object type	Meaning	Used for
`int`	integer value	natural numbers
`float`	floating-point number	real numbers
`bool`	boolean value	true or false
`str`	string object	character, word, text

use built-in function `type()` to obtain the information

1.1 Integers and Floats

Integers are whole numbers, while **floats** are numbers with decimal values.

Int

In [1]:

```
a = 10
type(a)
```

Out[1]:

```
int
```

Arithmetic operations: `+` `-` `*` `/`

In [2]:
```
1 + 4
```

Out[2]:
```
5
```

In [3]:
```
a + 1
```

Out[3]:
```
11
```

In [4]:
```
type(1+4)
```

Out[4]:
```
int
```

Floats

In [5]:
```
type (1/4)
```

Out[5]:
```
float
```

In [6]:
```
1/4
```

Out[6]:
```
0.25
```

In [7]:
```
type(0.25)
```

Out[7]:
```
float
```

In [8]:
```
type (0)
```

Out[8]:
```
int
```

In [9]:

```
type (0.0)
```

Out[9]:

```
float
```

In [10]:

```python
# Example: Representing account balances
balance = 1000  # Integer
interest_rate = 5.5  # Float
```

In [11]:

```python
# Calculating interest
interest = balance * interest_rate / 100
print("Interest:", interest)
```

```
Interest: 55.0
```

1.2 Booleans

Booleans represent `True` or `False` values.

In [12]:

```python
# Example: Checking if an account is active
account_active = True
if account_active:
    print("The account is active.")
else:
    print("The account is inactive.")
```

```
The account is active.
```

In [13]:

```python
# implicit comparison
if account_active == True:
    print("The account is active.")
else:
    print("The account is inactive.")
```

```
The account is active.
```

Conditions: `>` `<` `>=` `<=` `==` `!=`

In [14]:
```
4 > 3
```

Out[14]:
```
True
```

In [15]:
```
type (4 > 3)
```

Out[15]:
```
bool
```

In [16]:
```
type (False)
```

Out[16]:
```
bool
```

In [17]:
```
4 >= 3
```

Out[17]:
```
True
```

In [18]:
```
4 < 3
```

Out[18]:
```
False
```

In [19]:
```
4 == 3
```

Out[19]:
```
False
```

In [20]:
```
4 != 3
```

Out[20]:
```
True
```

Logic operations: `and` `or` `not` `in`

In [21]:
```
True and True
```

Out[21]:
```
True
```

In [22]:
```
False and False
```

Out[22]:
```
False
```

In [23]:
```
True or True
```

Out[23]:
```
True
```

In [24]:
```
True or False
```

Out[24]:
```
True
```

In [25]:
```
False or False
```

Out[25]:
```
False
```

In [26]:
```
not True
```

Out[26]:
```
False
```

In [27]:
```
not False
```

Out[27]:
```
True
```

Combinations

In [28]:
```
(4 > 3) and (2 > 3)
```

Out[28]:
```
False
```

In [29]:
```
(4==3) or (2 != 3)
```

Out[29]:
```
True
```

In [30]:
```
not (4 != 4)
```

Out[30]:
```
True
```

In [31]:
```
(not (4 != 4)) and (2 == 3)
```

Out[31]:
```
False
```

Note: Major for control condition (`if` `while` `for`) -- see later

In [32]:

```
if 4 > 3:
    print ('condition true')
else:
    print ('condition not true')
```

```
condition true
```

In [33]:

```
i = 0
while i < 4:
    print ('condition true: i = ', i)
    i = i + 1
```

```
condition true: i =  0
condition true: i =  1
condition true: i =  2
condition true: i =  3
```

Boolean casting: 0,1 (and other values)

In [34]:
```
int(True)
```

Out[34]:
```
1
```

In [35]:
```
int(False)
```

Out[35]:
```
0
```

In [36]:
```
float(True)
```

Out[36]:
```
1.0
```

In [37]:
```
float(False)
```

Out[37]:
```
0.0
```

In [38]:
```
bool(0)
```

Out[38]:
```
False
```

In [39]:
```
bool(1)
```

Out[39]:
```
True
```

In [40]:
```
bool(0.0)
```

Out[40]:
```
False
```

In [41]:
```
bool(1.0)
```

Out[41]:
```
True
```

In [42]:
```
bool(10.5)
```

Out[42]:

```
True
```

In [43]:

```
bool(-2)
```

Out[43]:

```
True
```

1.3 Strings

Strings are used to represent text.

In [44]:

```
# Example: Representing account holder information
account_holder = "John Doe"
account_number = "1234567890"

print("Account Holder:", account_holder)
print("Account Number:", account_number)
```

```
Account Holder: John Doe
Account Number: 1234567890
```

In [45]:

```
type(account_holder)
```

Out[45]:

```
str
```

Built-in methods

`str` variables come with a series of useful built-in methods.

Method
`capitalize()`
`count()`
`find()`
`join()`
`replace()`
`split()`
`upper()`

In [46]:
```
t = 'this is a string object'
```

In [47]:
```
t.capitalize()
```

Out[47]:
```
'This is a string object'
```

In [48]:
```
t.split()
```

Out[48]:
```
['this', 'is', 'a', 'string', 'object']
```

In [49]:
```
t.find('string')
```

Out[49]:
```
10
```

In [50]:
```
t.replace(' ','|')
```

Out[50]:

```
'this|is|a|string|object'
```

Print method `print()`

In [51]:

```
print('Hello World!')
```

```
Hello World!
```

In [52]:

```
print (t)
```

```
this is a string object
```

In [53]:

```
i = 0
while i < 4:
    print (i)
    i = i + 1
```

```
0
1
2
3
```

In [54]:

```
i = 0
while i < 4:
    print (i, end = '|')
    i = i + 1
```

```
0|1|2|3|
```

Printing with variables

In [55]:

```
a = 10
print('this is the value of a:', a)
```

```
this is the value of a: 10
```

In [56]:

```
tt = 'this is the value of a: ' + str(a)
print (tt)
```

```
this is the value of a: 10
```

2. Basic structures

List of structures

Object type	Meaning	Used for
`tuple`	immutable container	fixed set of objects
`list`	mutable container	ordered and changing set of objects
`dict`	mutable container	key-value store
`set`	mutable container	unordered collection of unique objects

use built-in function `type()` to obtain the information

Navigating structures

- **Indexing**: obtain item at position *n*

```
s[n]
```

- **Slicing**: obtain items between position *i* and *j*

```
s[i:j]
s[i:]
s[:j]
```

- **Ranging**: obtain items between position *i* and *j* spaced by *k*

```
s[i:j:k]
```

Note: In Python, indexing starts at `0`

2.1 `tuple`

Tuples are **immutable** collections of items (i.e., cannot be changed after creation).

In [57]:

```
# Example: Coordinates of a bank branch
branch_location = (40.7128, -74.0060)  # New York City coordinates
print("Branch Location:", branch_location)
```

```
Branch Location: (40.7128, -74.006)
```

In [58]:
```
t = (1, 2.5, 'data')
type(t)
```

Out[58]:
```
tuple
```

In [59]:
```
#also works without ()
t = 1, 2.5, 'data'
type(t)
```

Out[59]:
```
tuple
```

In [60]:
```
#indexing
t[2]
```

Out[60]:
```
'data'
```

In [61]:
```
type(t[2])
```

Out[61]:
```
str
```

2.2 `list`

Lists are **ordered** collections of items, which can be of mixed data types.

In [62]:

```python
# Example: List of recent transactions
transactions = [100, -50, 200, -30, 400]
print("Transactions:", transactions)

# Adding a new transaction
transactions.append(-100)
print("Updated Transactions:", transactions)
```

```
Transactions: [100, -50, 200, -30, 400]
Updated Transactions: [100, -50, 200, -30, 400, -100]
```

In [63]:

```
l = [1, 2.5, 'data']
l[2]
```

Out[63]:

```
'data'
```

In [64]:

```
#casting
l = list(t)
l
```

Out[64]:

```
[1, 2.5, 'data']
```

In [65]:

```
type (l)
```

Out[65]:

```
list
```

Built-in methods

Method
`l[i] = x`
`l[i:j:k] = s`
`append()`
`count()`
`del l[i:j:k]`
`index()`
`extend()`
`insert()`
`remove()`
`pop()`
`revers()`
`sort()`

contrary to tuples, lists are mutable containers

In [66]:

```
l.append([4,3])
l
```

Out[66]:

```
[1, 2.5, 'data', [4, 3]]
```

In [67]:

```
l.extend([1.0, 1.5, 2.0])
l
```

Out[67]:

```
[1, 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]
```

In [68]:

```
l = [0, 1, 2, 3, 4, 5, 6, 7]
s = [10, 20, 30]

l[1:7:2] = s
print(l)
```

```
[0, 10, 2, 20, 4, 30, 6, 7]
```

In [69]:

```
l.insert(1,'insert')
l
```

Out[69]:

```
[0, 'insert', 10, 2, 20, 4, 30, 6, 7]
```

In [70]:

```
l.remove('data')
l
```

```
---------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [70], in <cell line: 1>()
----> 1 l.remove('data')
      2 l

ValueError: list.remove(x): x not in list
```

In [71]:

```
p = l.pop(3)
print (l, p)
```

```
[0, 'insert', 10, 20, 4, 30, 6, 7] 2
```

In [72]:

```
#slicing
l[2:5]
```

Out[72]:

```
[10, 20, 4]
```

2.3 `dict`

Dictionaries store data as key-value pairs.

In [73]:

```python
# Example: Dictionary of account balances
account_balances = {
    "1234567890": 1000,
    "0987654321": 2500,
    "1122334455": 750
}
print("Account Balances:", account_balances)

# Accessing a balance by account number
print("Balance of account 1234567890:", account_balances["1234567890"])
```

```
Account Balances: {'1234567890': 1000, '0987654321': 25
00, '1122334455': 750}
Balance of account 1234567890: 1000
```

Keys and values

In [74]:
```
d = {
    'Name' : 'Iron Man',
    'Country' : 'USA',
    'Profession' : 'Super Hero',
    'Age' : 36
}
```

In [75]:
```
type(d)
```

Out[75]:
```
dict
```

In [76]:
```
print (d['Name'], d['Age'])
```

```
Iron Man 36
```

Built-in methods

Method
`d[k]`
`d[k] = x`
`del d[k]`
`clear()`
`copy()`
`items()`
`keys()`
`values()`
`popitem()`
`update()`

In [77]:
```
d.keys()
```

Out[77]:
```
dict_keys(['Name', 'Country', 'Profession', 'Age'])
```

In [78]:
```
d.values()
```

Out[78]:
```
dict_values(['Iron Man', 'USA', 'Super Hero', 36])
```

In [79]:
```
d.items()
```

Out[79]:
```
dict_items([('Name', 'Iron Man'), ('Country', 'USA'),
('Profession', 'Super Hero'), ('Age', 36)])
```

In [80]:
```
birthday = True
if birthday:
    d['Age'] += 1
print (d['Age'])
```

```
37
```

In [81]:

```
for item in d.items():
    print (item)
```

```
('Name', 'Iron Man')
('Country', 'USA')
('Profession', 'Super Hero')
('Age', 37)
```

In [82]:

```
for value in d.values():
    print (type(value))
```

```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'int'>
```

2.4 `set`

Sets are unordered collections of unique items.

In [83]:

```
s = set(['u', 'd', 'ud', 'du', 'd', 'du'])
s
```

Out[83]:

```
{'d', 'du', 'u', 'ud'}
```

Set operations

In [84]:
```
t = set(['d', 'dd', 'uu', 'u'])
```

In [85]:
```
s.union(t)
```

Out[85]:
```
{'d', 'dd', 'du', 'u', 'ud', 'uu'}
```

In [86]:
```
s.intersection(t)
```

Out[86]:
```
{'d', 'u'}
```

In [87]:
```
s.difference(t)
```

Out[87]:
```
{'du', 'ud'}
```

In [88]:
```
t.difference(s)
```

Out[88]:

```
{'dd', 'uu'}
```

In [89]:

```
s.symmetric_difference(t)
```

Out[89]:

```
{'dd', 'du', 'ud', 'uu'}
```