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 [ ]:

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

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

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

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

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

Floats

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

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

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

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

In [ ]:
```
type (0.0)
```

In [ ]:
```
# Example: Representing account balances
balance = 1000  # Integer
interest_rate = 5.5  # Float
```

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

1.2 Booleans

Booleans represent `True` or `False` values.

In [ ]:

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

In [ ]:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Combinations

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

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

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

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

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

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

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

Boolean casting: 0,1 (and other values)

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

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

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

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

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

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

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

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

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

In [ ]:
```
bool(-2)
```

1.3 Strings

Strings are used to represent text.

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

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

In [ ]:
```
type(account_holder)
```

Built-in methods

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

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

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

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

In [ ]:
```
t.split('i')
```

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

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

Print method `print()`

In [ ]:
```
print('Hello World!')
```

In [ ]:
```
print (t)
```

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

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

Printing with variables

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

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

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 [ ]:

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

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

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

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

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

2.2 `list`

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

In [ ]:

```
# 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)
```

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

In [ ]:
```
#casting
l = list(t)
l
```

In [ ]:
```
type (l)
```

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

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

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

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

In [ ]:
```
l.insert(1,'insert')
l
```

In [ ]:
```
l.remove('data')
l
```

In [ ]:
```
p = l.pop(3)
print (l, p)
```

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

Mutable vs immutable objects

In [ ]:
```
t = (1, [2, 3], 4)
```

In [ ]:
```
t[1].append(5)
```

In [ ]:
```
print(t)
```

In [ ]:
```
t[0] = 9
```

2.3 `dict`

Dictionaries store data as key-value pairs.

In [ ]:

```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"])
```

Keys and values

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

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

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

Built-in methods

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

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

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

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

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

In [ ]:
```
for item in d.items():
    print (item)
```

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

2.4 `set`

Sets are unordered collections of unique items.

In [ ]:

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

Set operations

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

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

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

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

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

In [ ]:
```
s.symmetric_difference(t)
```