§1 Data Type

†a Object

Python is an **object-oriented** programing language. Everything is an **object** in Python:

$$\text{object} = \begin{cases} \text{identity}, \\ \text{type / class}, \\ \text{value / state}, \\ \text{methods / behaviors / operations}. \end{cases}$$

```
# print the identity, type, and the value for 4
print(id(4),type(4),4)
# type of any type is a type, the type itself is a type
print(type(type(4)))
print(type(type(type(4))))
```

```
140711773227544 <class 'int'> 4
<class 'type'>
<class 'type'>
```

- **Identity**: it guarantees that different objects have distinct identities at any given time.
- **Type**: objects of the same type support the same operations, and share the same properties.

†b Binding and Input

In Python, the **assignment** of $a = b$ is like making the name a pointing to the object b.

```
# an example for binding
a,b=4,print
print(type(a),a,type(b),b,
      id(a),id(4),id(b),id(print))
b(a+5,"hello")
```

```
<class 'int'> 4 <class 'builtin_function_or_method'>
    <built-in function print> 140723891816984
    140723891816984 2069908885472 2069908885472
9 hello
```

The basic input in Python is through the function input( ). The input takes ONE string as prompt, and it reads input as a string.

```
# an example for input function
n=input(f"{a} and hello\n")
print(type(n),n)
```

```
4 and hello
5
<class 'str'> 5
```

†c Numeric

The following are numeric types:

$$\text{bool} \subset \text{int} \subset_? \text{float} \subset_? \text{complex}^1$$

```
# an example for the above data types
print(type(True),True,type(1),1,
      type(1.0),1.0,type(1+0j),1+0j)
```

```
<class 'bool'> True <class 'int'> 1 <class 'float'> 1.0
    <class 'complex'> (1+0j)
```

```
# subset example
if True==1==1.0==1+0j:
   print("Yes")
else:
   print("No")
```

```
Yes
```

We can use bool( ), int( ), float( ), and complex( ) to convert a string to the corresponding data type from input( );

```
# input string to number
n=input("type in an integer\n")
print(type(n),n,type(int(n)),int(n))
```

```
type in an integer
17
<class 'str'> 17 <class 'int'> 17
```

identically map from a subset to a larger set, or canonically map from the supset to the restricted set:

```
# identical map and canonical map
print(int(False),float(5),int(3.7))
```

```
0 5.0 3
```

[1] In fact, int is not a subset of float nor complex, neither do float is a subset of complex.

More on Bool

```
# logic and bool
print(type(1==0))
print(type(""),bool(""))
if not "":
    print("statement or bool value defined can be used in
        logic")
```

```
<class 'bool'>
<class 'str'> False
statement or bool value defined can be used in logic
```

For statements and numbers, there is a **special method** bool:

```
# special method __bool__()
print(type((5==3).__bool__()),(5==3).__bool__(),
      id((5==3).__bool__()),id(False))
```

```
<class 'bool'> False 140723890821168 140723890821168
```

```
# special method __bool__() for numbers
print(type((0+3.5j).__bool__()),(0+3.5j).__bool__(),
      id((0+3.5j).__bool__()),id(True))
```

```
<class 'bool'> True 140723890821136 140723890821136
```

For some data like string, list etc., the special method len is defined.

```
# sepcial method __len__() for strings
print(type("abcd".__len__()),"abcd".__len__(),
      id("abcd".__len__()),id(4))
```

```
<class 'int'> 4 140723891816984 140723891816984
```

The bool( ) has a **protocol**:

1) If special method bool is defined, then return.
2) Else if the special method is defined, then return True if len is not 0 and vise versa.
3) Else return True.

In general, bool( ) is used for logical determination.

More on Float

For float, there are some useful **regular methods**:

```
# regular method is_integer() for floats
print(type((1.3).is_integer()),(1.3).is_integer())
```

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

```
# regular method as_integer_ratio() for floats
print(type((0.5).as_integer_ratio()),
      (0.5).as_integer_ratio())
```

```
<class 'tuple'> (1, 2)
```

In fact, for binary, 0.1 has infinitly many number after the decimal point:

```
# decimal in binary
x=0.1
print(x,f"{x:.17f}")
```

```
0.1 0.10000000000000001
```

For actual finite decimal, it is better to use a **standard library** decimal, and a function in the libary, Decimal:

```
# standard library decimal
import decimal
print(type(decimal.Decimal("0.1")),"\n",
      f"{decimal.Decimal("0.1"):.50f}\n",
      f"{decimal.Decimal(0.1):.50f}")
```

```
<class 'decimal.Decimal'>
 0.10000000000000000000000000000000000000000000000000
 0.10000000000000000555111512312578270211815834045410
```

A better way is to use a standard library fractions, and a fucntion in the libary, Fraction; and we can check if 0.1 is indeed the 0.1:

```
# standard library fractions
import fractions
print(type(fractions.Fraction(1,10)),"\n",
      fractions.Fraction(1,10),fractions.Fraction("0.1"),
      fractions.Fraction(1,10)==decimal.Decimal("0.1"))
```

```
<class 'fractions.Fraction'>
 1/10 1/10 True
```

†d String

Some regular methods: from string to string

```
# regular method strip() for string
n="    abc 123    "
print(n,"\n",n.strip())
```

```
    abc  123
 abc  123
```

```
# regular method lower(), upper() for string
n="AbCdEf"
print(n.lower(),n.upper())
```

```
abcdef ABCDEF
```

```
# regular method replace() for string
n="banana"
print(n.replace("a","A",2),n.replace("a","A"))
```

```
bAnAna bAnAnA
```

Regular method split: string to list with string elements, and join: **list with string elements** to string.

```
# regular method split(), join() for string
n="123.456.789"
print(type(n.split(".")),n.split("."),
      type(",".join(n.split("."))),",".join(n.split(".")))
```

```
<class 'list'> ['123', '456', '789'] <class 'str'>
    123,456,789
```

String is **immutable**: if we change the string, the identity is changed

```
# string is immutable
n="123"
m=n.strip("3")
print(n is m,n,m)
```

```
False 123 12
```

F-String

Formatted string literal, or f-string, is for better output.

```
# f-string examples
x=3.14159265358
print(f"Pi is approximately {x}\n",
      f"Pi is approximately {x:.4f}")
```

```
Pi is approximately 3.14159265358
 Pi is approximately 3.1416
```

In python, every float has at least 6 digits precision.

```
# precision
x=3.14159265358
print(f"{x:f}\n",
      f"{x:10f}")
```

```
3.141593
   3.141593
```

We can make the number show in different position, and fill the space with letter or number.

```
# integer
print(f"{int(x):5d}\n",
      f"{int(x):<5d}\n",
      f"{int(x):^5d}\n",
      f"{int(x):>5d}\n",
      f"{int(x):0<5d}\n",
      f"{int(x):e^5d}\n",
      f"{int(x):x>5d}")
```

```
    3
3
  3
    3
30000
ee3ee
xxxx3
```

†e List and Tuple

```
# list
a,b,c,d=[],[1],[1,],[1,2,3]
print(type(a),type(b),type(c),type(d),
      a,b,c,d)
```

```
<class 'list'> <class 'list'> <class 'list'> <class
    'list'> [] [1] [1] [1, 2, 3]
```

```
# tuple
a,b,c,d=(),(1),(1,),(1,2,3)
print(type(a),type(b),type(c),type(d),
```

```
    a,b,c,d)
```

```
<class 'tuple'> <class 'int'> <class 'tuple'> <class
    'tuple'> () 1 (1,) (1, 2, 3)
```

Contrast to string, list is **mutable**: we can change the list without changing the identity, and the method return None.

```
# regular method append() for list
a=[1,2]
b=a
c=a.append(3)
print(type(c),c,b,b is a)
```

```
<class 'NoneType'> None [1, 2, 3] True
```

The input for extend must be **iterable**.

```
# regular method extend() for list
a=[1,2]
b=[1,2]
print(a.extend([3,4]),b.append([3,4]),a,b)
```

```
None None [1, 2, 3, 4] [1, 2, [3, 4]]
```

The following require **indexing**.

```
# regular method insert(), pop(), index(), clear() for
    list
a=[1,3,4,5,5,6,7]
print(a.insert(1,2),a)
print(a.pop(),a,a.pop(0),a)
print(a.index(6))
try:
    print(a.index(7))
except ValueError:
    print("Value Error, cannot find the number 7 in a")
print(a.clear(),a)
```

```
None [1, 2, 3, 4, 5, 5, 6, 7]
7 [2, 3, 4, 5, 5, 6] 1 [2, 3, 4, 5, 5, 6]
5
Value Error, cannot find the number 7 in a
None []
```

We can search a specific element and remove the element:

```
# regular method count(), remove() for list
a=[1,2,3,4,5,5,5,5,5,5,5,6]
print(type(a.count(5)),a.count(5),a.remove(5),a,a.count(5))
for i in a:
    if a.count(i)>1:
        for _ in range(a.count(i)-1):
            a.remove(i)
print(a)
```

```
<class 'int'> 7 None [1, 2, 3, 4, 5, 5, 5, 5, 5, 5, 6] 6
[1, 2, 3, 4, 5, 6]
```

The difference between **shallow copy** and **deep copy**:

```
# regular method copy(), reverse() for list
a=[1,2,3,4]
b=a.copy()
print(a is b,a.reverse(),a,b)

# copy()'s elements still shared
a=[[1,2],[3,4]]
b=a.copy()
a.reverse()
print(a,b)
b[0][0]=999
print(a,b)

# deep copy
import copy
a=[[1,2],[3,4]]
b=copy.deepcopy(a)
a.reverse()
print(a,b)
b[0][0]=999
print(a,b)
```

```
[[3, 4], [1, 2]] [[1, 2], [3, 4]]
[[3, 4], [999, 2]] [[999, 2], [3, 4]]
[[3, 4], [1, 2]] [[1, 2], [3, 4]]
[[3, 4], [1, 2]] [[999, 2], [3, 4]]
```

While tuple is immutable, but the regular methods count, index are similar.