PYTHON CRASH COURSE: TYPES AND OPERATORS

By: Lekiz Rai

Date: 23/05/2024

Contents

Types

Python types

What is type?

In a programming language, a **type** is a **description** of a **set of values** and a **set of allowed operations** which are **denoted** by **operators** on those values.

Ex: In C++, we have types such as *int, float, double, string, char,...* along with their allowed operations like *adding, multiplying,...* which are denoted by operators '+', '*',...

Types in Python

In Python, there are many types that are provided. Here we list out only the most common types which are used widely in practice:

- **Boolean:** Denoted by *bool*
- **Integer:** Denoted by *int*
- **Floating-point:** Denoted by *float*
- **String:** Denoted by *str*
- **List:** Denoted by *list*
- **Dictionary:** Denoted by *dict*
- **Tuple:** Denoted by *tuple*
- **User-defined type:** Denoted by the name of that user-defined type

How to see type of one variable?

To see type of one variable, we can use the built-in function **type()**.

Note: To declare a variable of one type in Python, we just assign any variable to any value of that type.

Sample code

```
a = True
print(type(a)) # bool
a = 0
print(type(a)) # int
a = 2.0
print(type(a)) # float
a = "abc"
print(type(a)) # str
```

Sample code

```
a = [1, 2, 3]
print(type(a)) # list
a = {"a": 12, "b": 13}
print(type(a)) # dict
a = (1, 2, 3)
print(type(a)) # tuple
```

Boolean

Sample code

```
# Declare a boolean variable in Python
a = True
b = False
```

Integer

Sample code

```
# Declare an integer variable in Python
a = -123
b = 99999999999999999999
```

Note: In Python, very large number is not the concern; that means when we deal with Python, we do not need another type for very large value in integer. In contrast to Python, C++ needs both *int* and *long int* type to express integers.

Floating-point

Sample code

```
# Declare a float variable in Python
a = -1.234
b = 3e-999999999 # b is equal to 3x10^-999999999
c = .10234
```

Note: In Python, double-precision floating-point number is not the concern just the same as integer case above, *float* in Python includes *float* and *double* in C++.

String

Sample code

```
# Declare a string variable in Python
a = "ab\nc"
b = 'ab\nc'
c = """ab\nc"""
d = '''ab\nc'''
e = r"ab\nc"
f = f"ab\nc{c}"
```

Questions

What is the difference between **raw string** (r"") and **format string** (f"") in Python?

String indexing in Python

s =

0	1	2	3	4
a	**b**	**c**	**d**	**e**
-5	-4	-3	-2	-1

Given a string *s* = *"abcde"* in Python, character *"a"* of *s* is at index *0* or *-5* and we can access that character by using both those indexes.

Sample code

```
s = "abcde"
print(s[0]) # a
print(s[-5]) # a
```

List

List in Python

In Python, there is no type called **array**. Instead, it provides **list** type working like **array** type. The difference here is that **list** in Python is **not homogenously-typed**, that means any **diffent types** can be contained in the **same list**.

Sample code

```
# Declare a list variable in Python
a = [1, 2, 3]
b = [1, 2.0, "abc", True]
c = [1, -1.9, "abc", True, [1, 2, 3]]
```

List indexing in Python

lst =

0	1	2	3	4
1	**2.0**	**"abc"**	**True**	**[1, 2, 3]**
-5	-4	-3	-2	-1

Similar to string, given a list *lst = [1, 2.0, "abc", True, [1, 2, 3]]* in Python, element *[1, 2, 3]* of *lst* is at index *4* or *-1* and we can access that element by using both those indexes.

Sample code

```
lst = [1, 2.0, "abc", True, [1, 2, 3]]
print(lst[4]) # [1, 2, 3]
print(lst[-1]) # [1, 2, 3]
```

Dictionary

Dictionary in Python

In Python, **dictionary** type is a kind of **set of key-value pairs** type. That means each element in a dictionary is a **pair of key and value** and **each key** only maps to **one value**. **Any hashable types** can be used for **key** and **any types** can be used for **value**.

Note: In dictionary type, if we declare two elements having the same key value, only the last one is kept to ensure the one-to-one mapping property.

Sample code

```
# Declare a dictionary variable in Python
a = {"a": 1, "a": 2}
b = {1: "a", True: 3.0}
c = {"abc": True, -1.98: 45}
```

Dictionary indexing in Python

dct =

keys	"abc"	-1.98
values	**True**	**45**

Given a dictionary *dct = {"abc": True, -1.98: 45}* in Python, the value of *45* of *dct* corresponds to the key of *-1.98* and we can access the value by using that key.

Sample code

```
dct = {"abc": True, -1.98: 45}
print(dct[-1.98]) # 45
print(dct["abc"]) # True
```

Tuple

Tuple in Python

In Python, **tuple** type is **immutable list** type. That means we **cannot modify** a **tuple variable**. Beside immutability property, **all remained things** are the **same** as **list** type.

Note: If one element of a tuple is of mutable type, then we can modify that element but not the tuple.

Sample code

```
# Declare a tuple variable in Python
a = (1, -1.9, "abc", True, [1, 2, 3])
# Try to modify
a[0] = 3 # Error, tuple cannot be modified
a[4][0] = 3 # OK, the last element now is [4, 2, 3]
```

Operators

Operators

What is operator?

In programming, an **operator** is a **character** that **represents** a specific **mathematical** or **logical action** or **process** on its corresponding **operands**.

Note: There are many types of operators, the three most common types are *unary*, *binary* and *ternary* operator. What is the difference between them?

Operator summary

Note: Each expression has format as *<Opr 1> <Op> <Opr 2>* (if *<Opr 1>* is empty, the format will be *<Op> <Opr 2>*), the *Output* is the type after applying operator on corresponding operand(s).

Op	Opr 1	Opr 2	Output	Function
$+$	*int*	*int*	*int*	Adding two numbers
	int	*float*	*float*	
	float	*int*	*float*	
	float	*float*	*float*	
	str	*str*	*str*	Concatenating two strings
	list	*list*	*list*	Extending the first list witht the second
	tuple	*tuple*	*tuple*	
$-$	*Same as "+" operator on numbers*			Subtracting two numbers
		int	*int*	Taking the negative value of number
		float	*float*	

Note: The notation *numeric* here stands for both *int* and *float*.

Op	Opr 1	Opr 2	Output	Function
$*$	*Same as "+" operator on numbers*			Multiplying two numbers
	str	*int*	*str*	Duplicating the string n times
	list	*int*	*list*	Duplicating the list n times
	tuple	*int*	*tuple*	
/	*numeric*	*numeric*	*float*	Dividing two numbers
//	*Same as "+" operator on numbers*			Taking the value of $\left\lfloor \frac{<Opr1>}{<Opr2>} \right\rfloor$
%	*Same as "+" operator on numbers*			Doing modulo between two numbers
$**$	*Same as "+" operator on numbers*			Doing exponent $< Opr1 >^{<Opr2>}$

Note: Operators *or* and *and* can be used with any types that can be casted into *bool* type. In Python, short-circuit evaluaiton is used. In case of *not bool* type, the returned value of those operators is based on operands' types and short-circuit evaluation mechanism.

Op	**Opr 1**	**Opr 2**	**Output**	**Function**
|	*int*	*int*	*int*	Doing bitwise OR between two numbers
&	*int*	*int*	*int*	Doing bitwise AND between two numbers
∧	*int*	*int*	*int*	Doing bitwise XOR between two numbers
∼		*int*	*int*	Doing bitwise NOT on numbers
or	*bool*	*bool*	*bool*	Doing OR on booleans
and	*bool*	*bool*	*bool*	Doing AND on booleans
not		*bool*	*bool*	Doing NOT on boolean

Note: The notation *except dict* here stands for all types except dictionary, the notation *any* stands for all types.

Op	Opr 1	Opr 2	Output	Function
$>$	*except dict*	*except dict*	*bool*	Doing greater-than comparison
$<$	*except dict*	*except dict*	*bool*	Doing less-than comparison
$>=$	*except dict*	*except dict*	*bool*	Doing greater-or-equal comparison
$<=$	*except dict*	*except dict*	*bool*	Doing less-or-equal comparison
$!=$	*any*	*any*	*bool*	Doing not-equal comparison on value
$==$	*any*	*any*	*bool*	Doing equal comparison on value

Questions

What is the difference between *is* and $==$ also between *is not* and *!=*? (operators *is* and *is not* are mentioned in next slide)

Note: The notation *list-like* here stands for two list-like types *list* and *tuple*.

Op	Opr 1	Opr 2	Output	Function
in	*any*	*list-like*	*bool*	Searching if element inside list-type
is not	*any*	*any*	*bool*	Doing not-equal comparison on object
is	*any*	*any*	*bool*	Doing equal comparison on object

Ternary operator

Beside unary and binary operators, Python provides a **ternary operator** having format as <*val0*> **if** <*bool*> **else** <*val1*>. Previous expression will return *val0* if the expression *bool* is **true** or return *val1* if the expression *bool* is **false**.

Sample code

```
a = 1 if True else 2 # a has value 1
b = 1 if False else 2 # b has value 2
```

Other operations

Note: In this part if we have variable *x*, we can show all posible operations on it by typing *"x."* then press the button *Tab* on our keyboard.

Other operations

Other operations on list

There are many other operations on *list* type such as:

- **lst.append(val):** Appending *val* to the tail of *lst*.
- **lst.extend(lst0):** Extending to the tail *lst* with another list *lst0*.
- **lst.insert(idx, val):** Inserting *val* at the position *idx* of *lst*.
- **lst.remove(val):** Removing *val* in *lst*, if not exist it will raise error.
- **lst.reverse():** Reversing the *lst*.
- **lst.sort():** Sorting the *lst*.
- **lst.clear():** Clearing the *lst*.
- **lst.copy():** Creating and returning a copy version of *lst*.
- Other operations

Other operations on dictionary

There are many other operations on *dictionary* type such as:

- **dct.get(key):** Returning the value corresponding to *key* in *dct*.
- **dct.update({key0: val0, key1: val1,...}):** Updating the values of *key0-val0*, *key1-val1*,... pairs in *dct*, can be used to add or modify.
- **dct.pop(key):** Popping and returning the value corresponding to *key* in *dct*.
- **dct.popitem():** Popping and returning the last ordered item in *dct*.
- **dct.clear():** Clearing the *dct*.
- **dct.copy():** Creating and returning a copy version of *dct*.
- **dct.keys():** Returning list of key values in *dct*.
- **dct.values():** Returning list of value values in *dct*.
- Other operations

Other operations on tuple

There are many other operations on *tuple* type such as:

- **tup.index(val):** Returning the index value of *val* in *tup*, if not exist it will raise error.
- **tup.count(val):** Returning the number of *val* appearing in *tup*.

Short-hand operation

Short-hand operation on list

- **lst[idx]:** Returning the value at the position *idx* of *lst*.
- **lst[idx] = val:** Changing the value at the position *idx* of *lst* into *val*.

Short-hand operation on dictionary

- **dct[key]:** Returning the value corresponding to *key* in *dct*.
- **dct[key] = val:** Changing the value corresponding to *key* in *dct* into *val*, if *key* not exist then add new pair *key-val* into *dct*.

Short-hand operation on tuple

- **tup[idx]:** Returning the value at the position *idx* of *tup*.

Short-hand operation for assigning

Python provides a kind of short-hand programming *a += 1* as the representation of *a = a + 1*. This programming style supports many operators such as +, -, *, /, //, **,...

Sample code

```
a = 3
a += 1 # a now is 4
a *= 2 # a now is 8
a /= 4 # a now is 2.0
s = "abc"
s += "d" # s now is "abcd"
```

Slicing operation

Slicing operation for list and tuple

Given one list/tuple *lst*, operation format is *lst[<start>:<end>:<step>]*. **Slicing operation** will **return** a **slice** of *lst* which **start** at position *<start>*, **end** at position just before *<end>* and each selected element in *lst* is **seperated** by *<step>*.

Note: If *<step>* is empty, slicing operation will take the default value which is 1. Also, slicing operation will choose for us the compatible values for *<start>* and *<end>* if they are not provided, the most common two values for them are the very first position (not 0) and the very last position (not -1).

Sample code

```
# Below examples can be applied for both
# list and tuple
a = [1, 2, 3, 4]
b = a[:] # b = [1, 2, 3, 4]
c = a[:2] # c = [1, 2]
d = a[1:4] # d = [2, 3, 4]
e = a[::2] # e = [1, 3]
f = a[-1::-1] # f = [4, 3, 2, 1]
g = a[::-1] # g = [4, 3, 2, 1]
h = a[-1:0:-2] # h = [4, 2]
```

Note for type behaviour

Type conversion in Python

Types of type conversion

There are two types of type conversion:

- **Implicit conversion (coersion):** The **conversion job** is **done** by **translator** at **runtime**, this is the reason why one operator can be used with many types in Python. For example we have the statement $a = 3 + 2.0$, the value 3 is converted implicitly into *float* type as *3.0* to perform the adding operation.
- **Explicit conversion (casting):** The **conversion job** is **done** by **translator** at **translating time**, it is usually done by using built-in casting functions such as *int()*, *str()*,... For example we have the statement $a = int(4.0)$, now a will have the integer value of 4.

Copy in Python

Kinds of copy in Python

There two kinds of copy in Python:

- **Deep copy:** When we assign a variable to another variable, the **whole value** of **assigning variable** will be **copied** and then **assigned** to **assigned one**. Ex: *int*, *float*, *bool*,...
- **Shallow copy:** When we assign a variable to another variable, the **assigned variable** is actually a **reference** to **assigning one** or they have **the same address** value. Ex: *list*, *dictionary*, *tuple*,...

Note: We can use the built-in function *id()* to see the address value of one variable. For example, *print(id(a))* will print out the address value of variable *a*.

Shallow copy issues

What is the problem?

- In shallow copy case, any **changes** made on **assigned variable** will be **conducted** on **assigning variable** as well and **vice versa**.
- To solve this problem, Python provides operation **copy()** to create a **deep copy version** of **assigning variable**.

Sample code

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a
d = b.copy()
c[0] = 5 # Variable a now is [5, 2, 3]
d[0] = 5 # Variable b now is still [1, 2, 3]
```

Some tricky questions

Questions

1. What is the value of *3 and 2 and 4*?, what about *3 and 0 and 1*?
2. What is the value of *2 or 3 or 4*?, what about *7 or 0 or 2*?
3. What is the value of *True + True*?
4. What is the value of *not 3*?
5. What is the value of *[1, 2] and {"a": 13, -1.92: True}*? What about *(1, 2) or {"a": 13, -1.92: True}*?
6. What is the value of *[] and 3*? What about *[] or 3*?
7. What is the value of *not {}*? What about *not ()*?