

03_PHYTON (Data Types & Useful Operations)

Basic Data Types

Introduction to Data Types

Each data has a type, whether constant or variable. This type of data defines how you store it in memory and it also describes which process can be applied to it.

In fact, data types are nothing but variables you use to reserve some space in memory. Python variables do not need an explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable.

💡 **Tips:**

- Note that, we assign value to a variable using ➡ =

You can think of types in the real world, bees are bee type and palm trees are palm type. That is, they have certain formats and common features.

We will now discuss some simple data types commonly used in Python:

- String,
- Signed Integer,
- Floating Point,
- Complex,
- Boolean.

Strings

Strings are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pair of single or double (or even triple) quotes.

Strings are immutable sequence data type, i.e each time one makes any changes to a string, a completely new string object is created. You will be able to better understand immutability with examples that will continue in the next lessons.

If you want to work with any textual characters in your code, you have to work with strings. The string type is called `str`. Strings are the most common and useful data type in Python.

Carefully examine the following examples that will help you understand the type of string.

input :

```
text1 = "I have learned strings" # surrounded with double quotes
print(text1)
```

output :

```
I have learned strings
```

input :

```
e_mail = 'joseph@clarusway.com' # surrounded with single quotes
print(e_mail)
```

output :

```
joseph@clarusway.com
```

input :

```
print('632') # this is also a string type
```

output :

```
632
```

Numeric Types

For any programmer, using numbers is the most important issue. You can hardly write a serious program without using numbers, so let's talk about some basic numeric types. There are three distinct numeric types: signed integer numbers, floating point numbers and complex numbers.

- Signed Integer type is called `int`, they are whole numbers (positive, negative or zero), including no decimal point. For example: `71`, `-122`, `0`
- Floating point type is called `float` and they stand for real numbers with a decimal point. For example: `71.0`, `-33.03`
- Complex type is called `complex` and they are written in the form, $x + yj$, where x is the real part and y is the imaginary part. For example: `3.14j`. Imaginary numbers, also called complex numbers, are used in real-life applications, such as electricity, as well as quadratic equations. In quadratic planes, imaginary numbers show up in equations that don't touch the x -axis. Imaginary numbers become particularly useful in advanced calculus. We will not use this type much.

Tips:

- `71` and `71.0` have the same numerical value. But they differ in terms of numeric type. The types of these numbers are `int` and `float` respectively.

Q: What are the numerical data types in Python and their properties?

A:

- Integers : they are whole numbers (positive, negative or zero), including no decimal point.
- Floats : they stand for real numbers with a decimal point.
- Complexes : they are written in the form, $x + yj$, where x is the real part and y is the imaginary part.

- Interview Q&A

Boolean

Boolean types are called `bool` and their values are the two constant objects False and True. They are used to represent truth values (other values can also be considered false or true). In numeric contexts (for example, when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively.

Bools are important data types that are widely used in Python as they can find use in every aspect of our daily lives. For example, imagine, whether the TV is turned on or off in your home or if the weather is rainy can be explained easily with bools.

Bools are mostly used in conditional operations which we will discuss in the next lessons.

```
tv_open = True # it seems TV is on now
```

```
is_rainy = False # I love sunny weather
```

Q: Describe the Boolean types in detail.

A: Boolean types are called `bool` and their values are the two constant objects True and False. They are used to represent truth values (other values can also be considered false or true).

In numeric contexts (for example, when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively.

Bools are important data types that are widely used in Python as they can find use in every aspect of our daily lives. For example, imagine, whether the TV is turned on or off in your home or if the weather is rainy can be explained easily with bools.

- Interview Q&A

Type Conversion

We can convert the types of data to each other if the type allows to be converted. There are some functions to convert the types:

- `str()` converts to string type
- `int()` converts to signed integer type
- `float()` converts to floating point type



Tips:

- We can print the types of data using `type()` function.

Look at the examples below to how we learn the types of data.

input :

```
example1 = 'sometimes what you say is less important th  
an how you say it'  
print(type(example1))
```

output :

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

input :

```
example2 = '71'  
print(type(example2))
```

output :

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

input :

```
example3 = 71  
print(type(example3))
```

output :

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

input :

```
example4 = 71.0  
print(type(example4))
```

output :

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

input :

```
example5 = 3.14j  
print(type(example5))
```

output :

```
<class 'complex'>
```

input :

```
example6 = True
print(type(example6))
```

output :

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



Tips:

- Note that we write the first letter of **True** in uppercase. This is the rule of Python that we must write like this : **True**, **False**.

Here are some examples on converting between different types:

input :

```
f = 3.14 # the type is float
print(type(f))
```

output :

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

input :

```
f = 3.14 # the type is float

s = str(f) # converting float to string
print(type(s))
```

output :

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

input :

```
f = 3.14 # the type is float

i = int(f) # while converting a float value to an integer its decimal part is disregarded
print(i, '\n')
print(type(i))
```

output :

```
3

<class 'int'>
```

input :

```
i = 3

f = float(i)
print(f, '\n')
print(type(f))
```

output :

```
3.0

<class 'float'>
```

input :

```
x = 39
v = "11"
y = "2.5"
z = "I am at_"
```



```
print(x-int(v))  
print(x-float(y))  
print(z+str(x))
```

output :

```
28  
36.5  
I am at_39
```



Tips:

- Note that, it is important that the value of any type in Python can be converted to a string.

Q: What are the 'type conversion' and basic methods of that in Python?

A: Type conversion refers to the conversion of one data type into another.

int() – converts some data types into integer type.

float() – converts some data types into float type.

str() – converts any data type into string type.

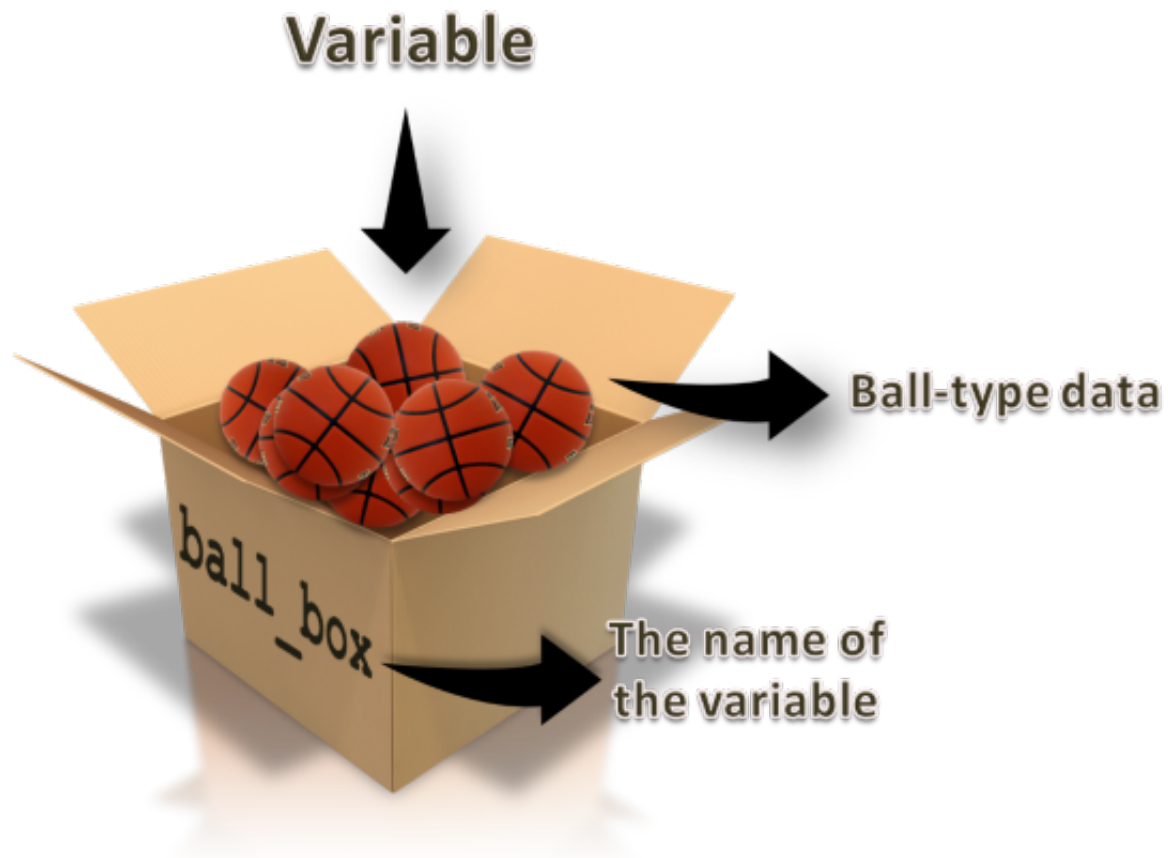
- Interview Q&A

Variables

Variable is a location designated where a value can be stored and accessed later. Imagine a box where you store something. That's a variable.

Let's create a box (variable) in which contains basketball balls. Let's name it **ball_box**. It is also the name of the variable.

Creating, naming the variable and assigning a value to it happen simultaneously by this syntax : `ball_box = 20 basketball balls`



Python variables do not need an explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable.

To create a variable in Python, all you need to do is specify the variable name and then assign a value to it using `=`

The formula syntax of creating a variable and assigning a value to it is :

```
variable name = value
```

Remember, according to the PEP8, we had to give the variables a meaningful name for the data they kept inside.

Let's define variables and assign values to them :

input :


```
color = 'red' # str type variable  
season = 'summer'
```

```
price = 250 # int type variable
pi = 3.14 # float type variable
color = 'blue' # You can always assign a new value to
a created variable
price = 100 # value of 'price' is changed
season = 'winter'

print(color, price, season, sep=', ')
```

output :

blue, 100, winter

 Scratch Time ! : Solve this example with [scratch](#).

 Tips:

- Note that, the last value assigned to a variable is valid.

In Python, it is possible to assign the value of one variable to another variable:

input :

```
a = 5
b = 55
c = 555
c = a
b = c
a = b

print(a, b, c, sep=', ')
```

output :

5, 5, 5

⚠️ Avoid ! :

- Note that, If you use undefined name of a variable in the code you write, you will get an '**NameError**' message.

Q: What is the 'variable' and how do you assign a value to it?

A: Variable is a location designated where a value can be stored and accessed later. Imagine a box where you store something. That's a variable.

Python variables do not need an explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable.

To create a variable in Python, all you need to do is specify the variable name and then assign a value to it.

- Interview Q&A

Simple Operations

Arithmetic Operations

In Python, there are almost all of the arithmetic operations we use in mathematics. They are so simple to use and we can also use these operations on almost all data types, including string.

Basic Arithmetic operators are as follows :

Arithmetic Operators in Python

Let's now grasp the arithmetic operations with several examples:

input :

```
print(4 + 11) # sum of integers gives integer
```

output :

15

input :

```
print(39 + 1.0) # sum of an integer and float gives float
```

output :

40.0

input :

```
no1, no2 = 46, 52
no3 = no1 - no2
print(no3)
```

output :

-6

💡 Tips:

- We can assign a value to multiple variables. Consider this: `variable1 = variable2 = 'clarusway opens your path'`.
 - We can also assign multiple values to multiple variables in sequence using commas as in example above.
-

input :

```
no1 = 46  
print(no1/23)  # division gives float
```

output :

```
2.0
```

input :

```
print((3 * 4)/2)  # parentheses are used as in normal m  
athematics operations
```

output :

```
6.0
```

input :

```
print(7 // 2)  # it gives integer part of division
```

output :

```
3
```

input :

```
print(9 % 2)  # remainder of this division is 1  
# it means 9 is an odd number
```

output :

```
1
```

input :

```
print(3**2)
```

output :

9

input :

```
print(2**3)
```

output :

8

input :

```
print(64**0.5) # square root
```

output :

8.0

input :

```
print('Result of this (12+7) sum :', 12 + 7)
```

output :

Result of this (12+7) sum : 19

There is a list of priorities for all considered operations: it is worth keeping this priority in your mind.

1. **parentheses : `()`**
2. **power : `**`**
3. **unary minus : `-3`**
4. **multiplication and division : `*`, `/`**
5. **addition and subtraction : `+`, `-`**

Operations with 'print()' Function

Let's open a title here and take a closer look at `print()`, which is the most frequently used function. Since the need to make constant changes and see the results frequently occurs when writing code, printing directly on the screen can be the choice of most developers.

input :

```
number = 2020
text = "children deserve respect as much as adults in"
print(text, number)
```

output :

```
children deserve respect as much as adults in 2020
```

When using `print()` we can write more than one expression in parentheses separated by `,`

input :

```
print("yesterday I ate", 2, "apples")
```

output :

yesterday I ate 2 apples

When you type more than one expression in `print()`, you notice that the expressions are joined to each other by spaces. This is due to the default value of keyword argument `sep` in the `print()` function. This argument, which is defined as a space `" "` by default, is not visible in the background in the `print()`.

The `print()` command automatically switches to the next line. This is due to the keyword argument `end = "\n"`

Here are the keyword arguments that run in the background of the `print()` function :

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

- `\n` represents next line.
 - Let's focus on the arguments `sep=` and `end=` we discussed above:
-

input :

```
print('i', end=' ')
print('will say', end=' ')
print("'i missed you'", end=' ')
print('to my mother')
```

output :

```
i will say 'i missed you' to my mother
```

If you noticed, in the example above we have combined all expressions with space in a single line using the `end=`. If we didn't use `end=' '`, we would normally get 4 lines of output.

input :

```
print('smoking', 'is', 'slowly', 'killing me', sep=' +
')
```

output :

```
smoking + is + slowly + killing me
```

Some other useful operations that covers arithmetic and `print()` function are as follows. Carefully examine the examples:

input :

```
x = 5
print ('value of x      : ', x)

x += 2
print ("2 more of x      : ", x, "\n") # using string e
expression '\n',
                                     # we produce extra
                                     # So that we had e

y = 10
print ('value of y      : ', y)

y -= 2
print ("2 minus y       : ", y, "\n")

z = 6
print ('value of z      : ', z)

z *= 2
print ("2 times z       : ", z, "\n")
```

output :

```
value of x      : 5
2 more of x     : 7

value of y      : 10
```

```
2 minus y      : 8
value of z     : 6
2 times z      : 12
```



Tips:

- **Variable math operator = number** gives the same result as **Variable = Variable math operator number**.
- **Variable += number** gives the same result as **Variable = Variable + number**.

Any mathematics operator can be used before the **=** character to make an in-place operation:

- **-=** decrements the variable in place,
 - **+=** increment the variable in place,
 - ***=** multiply the variable in place,
 - **/=** divide the variable in place,
 - **//=** floor divide the variable in place,
 - **%=** returns the modulus of the variable in place,
 - ****=** raise to power in place.
-

input :

```
fruit = 'Orange'
vegetable = "Tomato"
print (fruit, "" and "" , vegetable)
```

output :




```
Orange  and  Tomato
```



Tips:

- Remember. There is no difference between **' '**, **" "** or **""**.

Escape Sequences

Actually, the examples in the previous lesson show us how backslash  works.  is a special sign used in expressions called escape sequences, which behaves according to the character immediately after . Here are basic escape sequences in Python:

- `\n` : means new line,
- `\t` : means **tab** mark,
- `\b` : means backspace. It moves the cursor one character to the left.

Look at these examples carefully:

input :

```
print('C:\\november\\number_expenditure.txt')
```

output :

```
C:\november
umber_expenditure.txt
```

input :

```
print("one", "two", "three", sep="\t") # separated by tab marks
```

output :

```
one      two      three
```

input :

```
print('we', '\bare', '\bunited') # remember, normally p
rint() function
# separates expressions by spaces
```

output :

```
weareunited
```

Normally when we use `'` inside the `'`, Python will give error. Because single-quote in single quotes gives an error. But here, in the example below, `'\'` allows single-quote `'` to be ignored. So it gives no error.

input :

```
print('it\'s funny to learn Python')
```

output :

```
it's funny to learn Python
```

⚠️ Avoid ! :

- Be careful, when using `'\'` in the long string. It may cause error because of its functionality described above. Using `'\\'` guarantees no error.

Boolean Operations

Definitions

As we learned in the previous lesson boolean or `bool` can only have two values. `True` and `False`.

To put it easily, we can say that `bool` represent 1 and 0. In other words, yes & no or exist & nonexistent can be expressed by `bool` type.

For example, let's define a variable as to whether students have passed a course. Let the variable be called `is_pass`. Then;

If you pass the course : `is_pass = True`,
If you did not pass the course : `is_pass = False`

Q: What is a boolean in Python?

A: Boolean is one of the built-in data types in Python, it mainly contains two values, and they are True and False.

- Interview Q&A

Boolean Logic Expressions

Python has three built-in boolean operators: `and`, `or` and `not`. Except `not`, all are binary operators, which means two arguments are required.

And operator : The `and` operator evaluates all expressions and returns the last expression if all expressions are evaluated `True`. Otherwise, it returns the first value that evaluated `False`.

d

Or operator : The `or` operator evaluates the expressions left to right and returns the first value that evaluated `True` or the last value (if none is `True`).

Value1	Logic	Value2	Returns
True	and	True	True
True	and	False	False
False	and	False	False
False	and	True	False

True	or	True	True
True	or	False	True
False	or	False	False
False	or	True	True

Q: Python has three built-in Boolean operators. What are they?

A: They are **and**, **or**, **not**.

- Interview Q&A

Order of Priority

It is important to remember that, logical operators have a different priority and it has an effect on the order of evaluation. Here are the operators in order of their priorities:

1. **not**
2. **and**
3. **or**

For example : `x = True and not True`, the value of `x` returns **False**.

It evaluates `not True` first and gives **False**. It becomes `x = True and False` and gives **False**.

Let's consider one more example :

input :

```
logic = True and False or not False or False
print(logic)
```

output :

True



Tips:

- Note that **and** and **or** return one of its operands, not necessarily a **bool** type. But **not** always returns **bool** type.

Q: What is the order of priority of the logical operators?

A:

1. not
2. and
3. or

- Interview Q&A

Truth Values of Logic Statements

Although Python has its own boolean data type, we often use non-boolean values in logical operations.

The values of non-boolean types (integers, strings, etc.) are considered **truthy or **falsy** when used with logical operations, depending on whether they are seen as **True** or **False**.**

The following values are considered **False, in that they evaluate to **False** when applied to a boolean operator:**

- **None.**
- **Zero of any numeric type: 0, 0.0, 0j**
- **Empty sequences and collections: '', [], {}.**
- **Other than above values, any remaining value is evaluated as **True**.**

Here are some **and** operations :

input :

```
print(2 and 3)
```


output :

3

input :

`print(1 and 0)`

output :

0

input :

`print(1 and "I am doing good!")`

output :

I am doing good!

input :

`print([] and "Hello World!")`

output :

[]

Here are some **or** operations :

input :

```
print(2 or 3)
```

output :

2

input :

```
print(None or 1)
```

output :

1

input :

```
print(0 or {})
```

output :

{}

input :

```
print([] or "Hello World!")
```

output :

Hello World!

Q: What are the values evaluated to False when applied to a Boolean operator?

A:

- **None** and **False**.
- Zero of any numeric type: **0**, **0.0**, **0j**.
- Empty sequences and collections: **''**, **[]**, **{}**.
- Any remaining value is evaluated as **True**.

- Interview Q&A

The Strength of Strings in Python

Indexing&Slicing Strings

As we mentioned earlier, one of the most powerful aspects of Python is its string processing capability. You can access all elements of a string type data very easily. Accordance with the sequence of string letters, you can specify them from left to right in brackets, as follows:

input :

```
fruit = 'Orange'

print('Word           : ' , fruit)
print('First letter   : ' , fruit[0])
print('Second letter  : ' , fruit[1])
print("3rd to 5th letters : " , fruit[2:5])
print("Letter all after 3rd : " , fruit[2:])
```

output :

```
Word           : Orange
First letter    : 0
Second letter   : r
```

3rd to 5th letters : ang
Letter all after 3rd : ange

💡 Tips:

- Remember, the enumeration of a string starts from zero.

The formula syntax of string indexing is : **string[start:stop:step]**.

string[:] : returns the full copy of the sequence

string[start:] : returns elements from start to the end element

string[:stop] : returns element from the 1st element to stop-1

string[::step] : returns each element with a given step

Let's see it in an example :

input :

```
city = 'Phoenix'
```

```
print(city[1:]) # starts from index 1 to the end
print(city[:6]) # starts from zero to 5th index
print(city[::2]) # starts from zero to end by 2 step
print(city[1::2]) # starts from index 1 to the end by 2 step
print(city[-3:]) # starts from index -3 to the end
print(city[::-1]) # negative step starts from the end to zero
```

output :

```
hoenix
Phoeni
Ponx
hei
nix
xineohP
```

You can use the `len()` function to find out the length (number of characters) of a text or a variable of any type.

input :

```
vegetable = 'Tomato'

print('length of the word', vegetable, 'is :', len(vegetable))
```

output :

```
length of the word Tomato is : 6
```

Q: What is the output of `print(str[4:])` if `str = 'Python Language'` ?

A: `on Language`

- Interview Q&A

String Formatting with Arithmetic Syntax

There are several ways in Python that we use when processing and using string data structures. The most important of these are:

- **Arithmetic syntax (+, *, and =),**
- **% operator formatting,**
- **string.format() method,**
- **f-string formatting.**

We have stated to you what the function is? in the previous lessons. At this point, let us give the definition of the term method. A method is like a function, except it is attached to an object. We call a method on an object, and it possibly makes changes to that object (like `string.format()`). A method, then, belongs to a class.

Arithmetic syntax (+, =, *) :

We can use + operator for combining the two string together without any spaces. For example :

input :

```
print('clarus' + 'way')
```

output :

```
clarusway
```

We can also use * operator for repeating the string without any spaces. For example :

input :

```
print(3*'no way!')
```

output :

```
no way!no way!no way!
```

Examine the following example carefully :

input :

```
fruit = 'Orange'  
vegetable = 'Tomato'  
print("using + :", fruit + vegetable)  
print("using * :", 3 * fruit)
```

output :

```
using + : OrangeTomato
using * : OrangeOrangeOrange
```

As with numeric types, we can do addition operation in-place either with string type using 🖐️+=. Look at the examples below :

input :

```
fruit = 'orange'
fruit += ' apple'

print(fruit)
```

output :

```
orange apple
```

input :

```
fruit = 'orange'
fruit += ' apple'
fruit += ' banana'
fruit += ' apricot'

print(fruit)
```

output :

```
orange apple banana apricot
```

Q: There are several ways in Python that we use when processing and using string data structures. What are the most important of these:

A:

- Arithmetic syntax (+, * and =),
- % operator formatting,
- string.format() method,
- f-string formatting.

- Interview Q&A

String Formatting with '%' Operator

The other way that you will learn to format the strings is % operator. This one is not a frequently used way, but it's worth learning.

'%' operator formatting :

👉 % operator gets the values in order and prints them in order using several characters accordingly. Look at the example :

For now, we used only s, d and f characters to specify the data type in a string.

input :

```
phrase = 'I have %d %s and %.2f brothers' % (4, "children", 5)
print (phrase)
```

output :

```
I have 4 children and 5.00 brothers
```

Here in the example, the % operator first takes '4' and puts it in the first % operator, then takes 'children' secondly and puts it in the second % operator and finally takes '5' and puts it in the third % operator.



Tips:

- In the '%s' syntax : s stands for 'string'.

- In the '%.2f' syntax : **f** stands for 'float'. In this example 2 digits after point.
- In the '%d' syntax : **d** stands for 'numeric'.


If you want, you can limit the character numbers of the strings. Here is an example :

input :

```
sentence = "apologizing is a virtue"  
print("%.11s" % sentence) # we get first 11 characters  
of the string
```

output :

apologizing

You can also use variables with % operator to format the string. Let's look at the example :

input :

```
print('%(amount)d pounds of %(fruit)s left' % {'amount': 33, 'fruit': 'bananas'})
```

output :

33 pounds of bananas left

In this example, we used two variables which are **amount** and **fruit**. If you noticed, we assign values to variables in curly braces '{}'. This format is a dictionary type that you will learn in the next lessons.

Q: What is the output of `print('%5s' % x)` if `x = "HelloWorld!"` ?

A: Hello

- Interview Q&A

String Formatting with 'string.format()' method

You can make strings change depending on the value of a variable or an expression. The main methods of Python to format the output are :

'`string.format()`' method :

`string.format()` method is the improved form of `% operator` formatting.


As in this example below, the value of expression comes from `.format()` method in order. Curly braces  `{}` receives values from `.format()`.

input :

```
fruit = 'Orange'
vegetable = 'Tomato'
amount = 4
print('The amount of {} we bought is {} pounds'.format(
    fruit, amount))
```

output :

The amount of Orange we bought is 4 pounds

If you've written more variables than you need in the `.format()` method, the extra ones just will be ignored. Using keywords in  `{}` makes string more readable. For example:

input :

```
print('{state} is the most {adjective} state of the {country}'.format(state='California', country='USA', adjective='crowded'))
```

output :

```
California is the most crowded state of the USA
```



Tips:

- If you have noticed, we do not have to write the keywords in `.format()` method in order.

There is no limit in Python language! You can combine both positional and keyword arguments in the same `.format()` method :

At this point, let us give you some explanations : Positional arguments are arguments that can be called by their position in the function or method definition. Keyword arguments are arguments that can be called by their names.

input :

```
print('{0} is the most {adjective} state of the {country}'.format('California', country='USA', adjective='crowded'))
```

output :

```
California is the most crowded state of the USA
```

You can use the same variable in a string more than once if you need it. Also, you can select the objects by referring to their positions in brackets.

input :

```
print("{6} {0} {5} {3} {4} {1} {2}".format('have', 6, 'months', 'a job', 'in', 'found', 'I willl'))
```

output :

I will have found a job in 6 months

⚠️ Avoid ! :

- Be careful not to write keyword arguments before positional arguments.

Using `str.format()` method is much more readable and useful than using `%-operator` formatting in our codes, but `str.format()` method can still be too wordy if you are dealing with multiple parameters and longer strings. At this point, the `f-string` formatting which you will learn in the next lesson suffices.

String Formatting with 'f-string'

It is the easiest and useful formatting method of the strings.

'f-string' formatting :

It makes string formatting easier. This method was introduced in 2015 with Python 3.6.

`f-string` is the string syntax that is enclosed in quotes with a letter `f` at the beginning. Curly braces `{ }` that contain variable names or expressions are used to replace with their values.

Sample of a formula syntax is : `f"strings {variable1} {variable2} string {variable3}"`

Let's look at the example below on how the syntax is simple and readable.

input :

```
fruit = 'Orange'
vegetable = 'Tomato'
amount = 6
output = f"The amount of {fruit} and {vegetable} we bought are totally {amount} pounds"

print(output)
```

output :

```
The amount of Orange and Tomato we bought are totally 6 pounds
```

You can use all valid expressions, variables, and even methods in curly braces. Look at the examples:

input :

```
result = f"{4 * 5}"

print(result)
```

output :

```
20
```

input :

```
my_name = 'JOSEPH'
output = f"My name is {my_name.capitalize()}"

print(output)
```

output :

My name is Joseph


There is also a multiline **f-string** formatting style. Follow the example :

input :

```
name = "Joseph"
job = "teachers"
domain = "Data Science"
message = (
    f"Hi {name}. "
    f"You are one of the {job} "
    f"in the {domain} section."
)
print(message)
```

output :

```
Hi Joseph. You are one of the teachers in the Data Science section.
```

If you want to use multiple **f-string** formatting lines without parentheses, you will have the other option that you can use backslash  between lines.

input :

```
name = "Joseph"
job = "teachers"
domain = "Data Science"
message = f"Hi {name}. " \
    f"You are one of the {job} " \
    f"in the {domain} section."

print(message)
```

output :

Hi Joseph. You are one of the teachers in the Data Science section.

Q: If you want to use multiple 'f-string formatting' lines without parentheses, what will be the other option that you can use?

A: You can use backslashes 🖱️\ between f-lines.

- Interview Q&A

Main String Operations

Searching a String

To search patterns in a string there are two useful methods called **startswith()** and **endswith()** that search for the particular pattern in the immediate beginning or end of a string and return **True** if the expression is found. Here are some simple examples. Examine the basic syntax of those methods carefully.

input :

```
text = 'www.clarusway.com'
print(text.endswith('.com'))
print(text.startswith('http:'))
```

output :

```
True
False
```

input :

```
text = 'www.clarusway.com'
print(text.endswith('om'))
print(text.startswith('w'))
```

output :

True
True

These methods have optional arguments `start` and `end`. We can specify the search by adding arguments so that the area of search is delimited by `start` and `end` arguments.

💡 **Tips:**

- Remember! Characters of string count from left to right and start with zero.

The formula syntaxes are :

- `string.startswith(prefix[, start[, end]])`
- `string.endswith(suffix[, start[, end]])`

Look at the example below:

input :

```
email = "clarusway@clarusway.com is my e-mail address"
print(email.startswith("@", 9))
print(email.endswith("-", 10, 32))
```

output :

True
True

Q: What are the `string.startswith()` and `string.endswith()` method used for? Describe how?

A: To search patterns in a string there are two useful methods called `startswith()` and `endswith()` that search for the particular pattern in the immediate beginning or end of a string and return `True` if the expression is found.

Changing a String

The methods described below return the copy of the string with some changes made.

How does the following syntax work?

A string is given first (or the name of a variable that represents a string), then comes a period followed by the method name and parentheses in which arguments are listed.

The formula syntax is : `string.method()`

Let's examine some common and the most important methods of string changing :

- **`str.replace(old, new[, count])` replaces all occurrences of old with the new.**

The count argument is optional, and if the optional argument count is given, only the first count occurrences are replaced. **count**: Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

- **`str.swapcase()` converts upper case to lower case and vice versa.**
- **`str.capitalize()` changes the first character of the string to the upper case and the rest to the lower case.**
- **`str.upper()` converts all characters of the string to the upper case.**
- **`str.lower()` converts all characters of the string to the lower case.**
- **`str.title()` converts the first character of each word to upper case.**

Let's consolidate the subject through examples :

input :

```
sentence = "I live and work in Virginia"
print(sentence.upper())
print(sentence.lower())
print(sentence.swapcase())
print(sentence) # note that, source text is unchanged
```

output :

```
I LIVE AND WORK IN VIRGINIA
i live and work in virginia
i LIVE AND WORK IN vIRGINIA
I live and work in Virginia
```

If we assign the modified text to a new variable, we can have a new string.
Consider these :

input :

```
sentence = "I live and work in Virginia"
title_sentence = sentence.title()
print(title_sentence)

changed_sentence = sentence.replace("i", "+")
print(changed_sentence)

print(sentence) # note that, again source text is unchanged
```

output :

```
I Live And Work In Virginia
I l+ve and work +n V+rg+n+a
I live and work in Virginia
```

input :

```
sentence = "I live and work in Virginia"
swap_case = sentence.swapcase()
print(swap_case)
print(swap_case.capitalize()) # changes 'i' to upperca
se and
# the rest to lowercase
```

output :

```
i LIVE AND WORK IN vIRGINIA
I live and work in virginia
```

Q: `print("Actions speaks louder than words".upper().swapcase().capitalize())`, will this code work? If yes, what the output will be? Describe how?

A: Yes it works. The syntax is : `string.method()`. Changing the string using these methods returns string type again. The output is :

`Actions speaks louder than words`

Follow the additional examples below :

```
string.upper() # returns string type,
string.upper().lower() # also returns string type,
string.upper().lower().title() # returns string type again.
```

- Interview Q&A

Q: What does the `title()` method do in Python?

A: For answer [click here](#).

- Interview Q&A

Editing a String

The methods described below remove the trailing characters (i.e. characters from the right side). The default for the argument chars is also whitespace. If

the argument chars aren't specified, trailing whitespaces are removed.

■ The formula syntax is : `string.method()`

- `str.strip()` : removes all spaces (or specified characters) from both sides.
- `str.rstrip()` : removes spaces (or specified characters) from the right side.
- `str.lstrip()` : removes spaces (or specified characters) from the left side.

Now see the examples about how we implement these methods? :

input :

```
space_string = "    listen first    "
print(space_string.strip()) # removes all spaces from
both sides
```

output :

```
listen first
```

input :

```
source_string = "interoperability"
print(source_string.strip("yi"))
# removes trailing "y" or "i" or "yi" or "iy" from both
sides
```

output :

```
nteroperabilit
```

input :

```
source_string = "interoperability"  
print(source_string.lstrip("in"))  
# removes "i" or "n" or "in" or "ni" from the left side
```

output :

```
teroperability
```

input :

```
space_string = "    listen first    "  
print(space_string.rstrip()) # removes spaces from the  
right side
```

output :

```
listen first
```

input :

```
source_string = "interoperability"  
print(source_string.rstrip("yt"))  
# removes "y" or "t" or "yt" or "ty" from the right side
```

output :

```
interoperabili
```

As we said before, Python's string processing capability is very good. In this context, many other string processing and editing methods are available.