**Data types or Data structures:: strings, list, dictionaries**

A string is a data type in Python that's used to represent a piece of text. It's written between quotes, either double quotes or single quotes. A string can be as short as zero characters, usually called an empty string.

A less common operation is to multiply the string by a number, which multiplies the content of the string that many times like this.

print("example"\*3)

output:: exampleexampleexample

If you want to print out an individual character from string…

string=”aabbccddee”

print(string[2])

What if you want to print the last character of a string but you don't know how long it is? You can do that using negative indexes.

string="Random string with a lot of characters"  
print(string[-1])  
print(string[-3])

output::

s

e

Warning:: If you try to access an index that’s larger than the length of your string, you’ll get an IndexError. This is because you’re trying to access something that doesn't exist!

We can also access a slice of a string. A slice is the portion of a string that can contain more than one character, also sometimes called a substring. We do that by creating a range using a colon as a separator.

string="aabbccddee"  
print(string[2:5])

output:: bbc

N.B::The range we use when accessing a slice of a string works just like the one created by the range function. It includes the first number, but goes up to one less than the last number.

Different ways of creating substring::

string="aabbccddee"  
print(string[:5])  
print(string[2:])

output::

aabbc

bbccddee

What if you try to replace a character from the string 🡺

txt = "A kong string with a silly typo"  
txt[2]="l"

output::

"F:\Interest\Python\week 3\venv\Scripts\python.exe" "F:/Interest/Python/week 3/practice.py"

Traceback (most recent call last):

File "F:/Interest/Python/week 3/practice.py", line 150, in <module>

txt[2]="l"

TypeError: 'str' object does not support item assignment

Process finished with exit code 1

We get this kind of error message when we try to replace a character from the string because strings are immutable(i.e they can not be modified).(Lists are mutable) To resolve this problem we have to create a new string based on the old one

txt = "A kong string with a silly typo"  
new\_txt=txt[0:2]+"l"+txt[3:]  
print(new\_txt)

To find the index of a character or a substring 🡺

txt="cats & Dogs"  
print(txt.index("&"))  
print(txt.index("Dogs"))  
print(txt.index("s"))  
print(txt.index("x"))

output::

5

7

3

Traceback (most recent call last):

File "F:/Interest/Python/week 3/practice.py", line 153, in <module>

print(txt.index("x"))

ValueError: substring not found

The index method returns the index of the given substring, inside the string. In this case, we're using a method to get the index of a certain character. A method is a function associated with a specific class. That is, this is a function that applies to a variable. And we can call it by following the variable with a dot. If a substring is not in the string, then we will get an ValueError.

To determine whether a substring is included in the string or not 🡺

txt="cats & Dogs"  
print("Dragons" in txt)  
print("a" in txt)

output::

False

True

In Python, strings are immutable. This means that they can't be modified. So if we wanted to fix a typo in a string, we can't simply modify the wrong character. We would have to create a new string with the typo corrected. We can also assign a new value to the variable holding our string.

If we aren't sure what the index of our typo is, we can use the string method index to locate it and return the index. Let's imagine we have the string **"lions tigers and bears"** in the variable **animals**. We can locate the index that contains the letter **g** using animals.index("g"), which will return the index; in this case 8. We can also use substrings to locate the index where the substring begins. animals.index("bears") would return 17, since that’s the start of the substring. If there’s more than one match for a substring, the index method will return the first match. If we try to locate a substring that doesn't exist in the string, we’ll receive a **ValueError** explaining that the substring was not found.

We can avoid a ValueError by first checking if the substring exists in the string. This can be done using the **in** keyword. We saw this keyword earlier when we covered for loops. In this case, it's a conditional that will be either True or False. If the substring is found in the string, it will be True. If the substring is not found in the string, it will be False. Using our previous variable **animals**, we can do **"horses"** **in animals** to check if the substring "horses" is found in our variable. In this case, it would evaluate to False, since horses aren’t included in our example string. If we did **"tigers" in animals**, we'd get True, since this substring is contained in our string.

Different method that can be used in case of ***string class* ::**

***Index ,Upper, lower, strip, lstrip, rstrip, count, endswith, isnumeric, isaplha join, split, replace method***

List.index(“abc”)

“abc”.upper() = ABC

“ABC”.lower() = abc

“ yes “.strip() = “yes”

“ yes “.lstrip() = ”yes “

“ yes “.rstrip() = ” yes”

“ifthekher uddin Chowdhury”.count(“I”) = 2

“animals”.endswith(“als”) = True

“animals”.endswith(“las”) = False

“animals”.isnumeric() = False

“1234”.isnumeric() = True

“123”.isalpha()=False

“12asb”.isalpha()=False

“abc”.isalpha()=True

“ “.join([“This”, “is”, “a”, “phrase”, “joined”, “by”, “spaces”]) = This is a phrase joined by spaces

“…“.join([“This”, “is”, “a”, “phrase”, “joined”, “by”, “triple”, “dots”]) = This…is…a…phrase…joined…by…spaces

“This is another example”.split() =[‘This’, ’is’, ‘another’, ‘example’]

"Hello World".replace("H","h")=hello world

* string.split() / string.split(delimiter) Returns a list of substrings that were separated by whitespace / delimiter
* string.replace(old, new) Returns a new string where all occurrences of old have been replaced by new.
* delimiter.join(list of strings) Returns a new string with all the strings joined by the delimiter

In earlier, we saw that we can concatenate strings using the plus sign. The join method can also be used for concatenating.

We've covered a bunch of String class methods already, so let's keep building on those and run down some more advanced methods.

The method **count** can be used to return the number of times a substring appears in a string.

The **isnumeric** method can check if a string is composed of only numbers. If the string contains only numbers, this method will return True. We can use this to check if a string contains numbers before passing the string to the **int()** function to convert it to an integer, avoiding an error. Useful!

We took a look at string concatenation using the plus sign, earlier. We can also use the **join** method to concatenate strings. This method is called on a string that will be used to join a list of strings. The method takes a list of strings to be joined as a parameter, and returns a new string composed of each of the strings from our list joined using the initial string. For example, **" ".join(["This","is","a","sentence"])** would return the string **"This is a sentence"**.

The inverse of the join method is the **split** method. This allows us to split a string into a list of strings. By default, it splits by any whitespace characters. You can also split by any other characters by passing a parameter.

Python | format() function

**str.format()** is one of the *string formatting methods* in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.

**Using a Single Formatter :**

Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces **{ }**into a string and calling the str.format(). The value we wish to put into the placeholders and concatenate with the string passed as parameters into the format function.

***Syntax :******{ } .format(value)***

***Parameters :******(value) :****Can be an integer, floating point numeric constant, string, characters or even variables.*

***Returntype :****Returns a formatted string with the value passed as parameter in the placeholder position.*

**Code #1 :** Simple demonstration of format().

# Python3 program to demonstarte

# the str.format() method

# using format option in a simple string

print ("{}, A computer science portal for geeks.".format("GeeksforGeeks"))

# using format option for a

# value stored in a variable

str = "This article is written in {}"

print (str.format("Python"))

# formatting a string using a numeric constant

print ("Hello, I am {} years old !".format(18))

output:

GeeksforGeeks, A computer science portal for geeks.

This article is written in Python

Hello, I am 18 years old!

#### Using Multiple Formatters :

Multiple pairs of curly braces can be used while formatting the string. Let’s say if another variable substitution is needed in sentence, can be done by adding a second pair of curly braces and passing a second value into the method. Python will replace the placeholders by values in **order.**

***Syntax :****{ } { } .format(value1, value2)*

***Parameters :******(value1, value2) :****Can be integers, floating point numeric constants, strings, characters and even variables. Only difference is, the number of values passed as parameters in format() method must be equal to the number of placeholders created in the string.*

***Errors and Exceptions :******IndexError :****Occurs when string has an extra placeholder and we didn’t pass any value for it in the format() method. Python usually assigns the placeholders with default index in order like*0, 1, 2, 3….*to acces the values passed as parameters. So when it encounters a placeholder whose index doesn’t have any value passed inside as parameter, it throws IndexError.*

**Code #2 :**

# Python program demonstrating Index error

# Number of placeholders are four but

# there are only three values passed

# parameters in format function.

my\_string = "{}, is a {} {} science portal for {}"

print (my\_string.format("GeeksforGeeks", "computer", "geeks"))

output:

Traceback (most recent call last):

File "/home/949ca7b5b7e26575871639f03193d1b3.py", line 2, in

print (my\_string.format("GeeksforGeeks", "computer", "geeks"))

IndexError: tuple index out of range

**Code #3 :** Formatters with multiple place holders.

# Python program using multiple place

# holders to demonstrate str.format() method

# Multiple placeholders in format() function

my\_string = "{}, is a {} science portal for {}"

print (my\_string.format("GeeksforGeeks", "computer", "geeks"))

# different datatypes can be used in formatting

print ("Hi ! My name is {} and I am {} years old".format("User", 19))

# The values passed as parameters

# are replaced in order of their entry

print ("This is {} {} {} {}".format("one", "two", "three", "four"))

**Output :**

GeeksforGeeks, is a computer science portal for geeks

Hi! My name is User and I am 19 years old

This is one two three four

#### Formatters with Positional and Keyword Arguments :

When placeholders **{ }**are empty, Python will replace the values passed through str.format() in order.

The values that exist within the str.format() method are essentially **tuple data types**and each individual value contained in the tuple can be called by its index number, which starts with the index number 0. These index numbers can be passes into the curly braces that serve as the placeholders in the original string.

***Syntax :****{0} {1}.format(positional\_argument, keyword\_argument)*

***Parameters :****(positional\_argument, keyword\_argument)*

***Positional\_argument****can be integers, floating point numeric constants, strings, characters and even variables.****Keyword\_argument****is essentially a variable storing some value, which is passed as parameter.*

**Code #4 :**

|  |
| --- |
| # To demonstrate the use of formatters  # with positional key arguments.    # Positional arguments  # are placed in order  print("{0} love {1}!!".format("GeeksforGeeks",                                      "Geeks"))    # Reverse the index numbers with the  # parameters of the placeholders  print("{1} love {0}!!".format("GeeksforGeeks",                                      "Geeks"))      print("Every {} should know the use of {} {} programming and {}".format("programmer", "Open", "Source", "Operating Systems"))      # Use the index numbers of the  # values to change the order that  # they appear in the string  print("Every {3} should know the use of {2} {1} programming and {0}".format("programmer", "Open", "Source", "Operating Systems"))      # Keyword arguments are called  # by their keyword name  print("{gfg} is a {0} science portal for {1}".format("computer", "geeks", gfg ="GeeksforGeeks")) |

**Output :**

GeeksforGeeks love Geeks!!

Geeks love GeeksforGeeks!!

Every programmer should know the use of Open Source programming and Operating Systems

Every Operating Systems should know the use of Source Open programming and programmer

GeeksforGeeks is a computer science portal for geeks

**Type Specifying :**

More parameters can be included within the curly braces of our syntax. Use the format code syntax **{field\_name:conversion}**, where field\_name specifies the index number of the argument to the str.format() method, and conversion refers to the conversion code of the data type.

*s – strings  
d – decimal integers (base-10)  
f – floating point display  
c – character  
b – binary  
o – octal  
x – hexadecimal with lowercase letters after 9  
X – hexadecimal with uppercase letters after 9  
e – exponent notation*

***Syntax :*** *String {field\_name:conversion} Example.format(value)*

***Errors and Exceptions :******ValueError :****Error occurs during type conversion in this method.*

**Code #5 :**

|  |
| --- |
| # Demonstrate ValueError while  # doing forced type-conversions    # When explicitly converted floating point  # values to decimal with base-10 by 'd'  # type conversion we encounter Value-Error.  print("The temperature today is {0:d} degrees outside !"  .format(35.567))    # Instead write this to avoid value-errors  ''' print("The temperature today is {0:.0f} degrees outside !"   .format(35.567))''' |

**Output :**

Traceback (most recent call last):

File "/home/9daca03d1c7a94e7fb5fb326dcb6d242.py", line 5, in

print("The temperature today is {0:d} degrees outside!".format(35.567))

ValueError: Unknown format code 'd' for object of type 'float'

**Code #6 :**

|  |
| --- |
| # Convert base-10 decimal integers  # to floating point numeric constants  print ("This site is {0:f}% securely {1}!!". format(100, "encrypted"))    # To limit the precision  print ("My average of this {0} was {1:.2f}%".format("semester", 78.234876))    # For no decimal places  print ("My average of this {0} was {1:.0f}%".format("semester", 78.234876))    # Convert an integer to its binary or  # with other different converted bases.  print("The {0} of 100 is {1:b}".format("binary", 100))    print("The {0} of 100 is {1:o}".format("octal", 100)) |

**Output :**

This site is 100.000000% securely encrypted!!

My average of this semester was 78.23%

My average of this semester was 78%

The binary of 100 is 1100100

The octal of 100 is 144

**Padding Substitutions or Generating Spaces :**

**Code #7 :**

By default strings are left-justified within the field, and numbers are right-justified. We can modify this by placing an alignment code just following the colon.

**<** : left-align text in the field

**^** : center text in the field

**>** : right-align text in the field

|  |
| --- |
| # To demonstrate spacing when  # strings are passed as parameters  print("{0:4}, is the computer science portal for {1:8}!"                          .format("GeeksforGeeks", "geeks"))    # To demonstrate spacing when numeric  # constants are passed as parameters.  print("It is {0:5} degrees outside !".format(40))    # To demonstrate both string and numeric  # constants passed as parameters  print("{0:4} was founded in {1:16}!".format("GeeksforGeeks", 2009))      # To demonstrate aligning of spaces  print("{0:^16} was founded in {1:<4}!".format("GeeksforGeeks", 2009))    print("{:\*^20s}".format("Geeks")) |

**Output :**

GeeksforGeeks, is the computer science portal for geeks !

It is 40 degrees outside!

GeeksforGeeks was founded in 2009!

GeeksforGeeks was founded in 2009 !

\*\*\*\*\*\*\*Geeks\*\*\*\*\*\*\*\*

### **Formatting expressions**

| **Expr** | **Meaning** | **Example** |
| --- | --- | --- |
| {:d} | integer value | '{:d}'.format(10.5) → '10' |
| {:.2f} | floating point with that many decimals | '{:.2f}'.format(0.5) → '0.50' |
| {:.2s} | string with that many characters | '{:.2s}'.format('Python') → 'Py' |
| {:<6s} | string aligned to the left that many spaces | '{:<6s}'.format('Py') → 'Py    ' |
| {:>6s} | string aligned to the right that many spaces | '{:>6s}'.format('Py') → '    Py' |
| {:^6s} | string centered in that many spaces | '{:^6s}'.format('Py') → '  Py ' |

### **Formatted string literals (Optional)**

This feature was added in Python 3.6 and isn’t used a lot yet.

A formatted string literal or f-string is a string that starts with 'f' or 'F' before the quotes. These strings might contain {} placeholders using expressions like the ones used for format method strings.

The important difference with the format method is that it takes the value of the variables from the current context, instead of taking the values from parameters. While other string literals always have a constant value, formatted strings are really expressions evaluated at run time

Examples: (Everything works like format() method, just syntax is different)

>>> name = "Micah"

>>> print(f'Hello {name}')

Hello Micah

>>> item = "Purple Cup"

>>> amount = 5

>>> price = amount \* 3.25

>>> print(f'Item: {item} - Amount: {amount} - Price: {price:.2f}')

Item: Purple Cup - Amount: 5 - Price: 16.25

Official documentation for [the format string syntax](https://docs.python.org/3/library/string.html#formatstrings)

<https://docs.python.org/3/library/string.html#formatstrings>

Check out the official documentation for [old string formatting](https://docs.python.org/3/library/stdtypes.html#old-string-formatting).

<https://docs.python.org/3/library/stdtypes.html#old-string-formatting>

Check out the official documentation for [f-strings](https://docs.python.org/3/reference/lexical_analysis.html#f-strings).

<https://docs.python.org/3/reference/lexical_analysis.html#f-strings>

