String Input and Output, Built-in Function Module

Topic 03 Key concept

1. String

1.1 Single Quotes and Double Quotes

- We learned that the characters between an opening quote and a closing quote are called a string, and we tried printing string using the print statement.
- We learned that string can be created using either double quotes (" ") or single quotes (' '). "Hello World!" is a string, and 'Hello World!' is also a string.

```
1 print('Hello World!')
2 print("Hello World!")
```

```
Hello World!
Hello World!
```

1. String

1.1 Single Quotes and Double Quotes

```
1 print('Hello World!')
2 print("Hello World!")
```

```
Hello World!
Hello World!
```

Line 1, 2

- When printing String, the same result is obtained whether we use single quotes like 'Hello Python!!' or double quotes like "Hello Python!!".

1. String

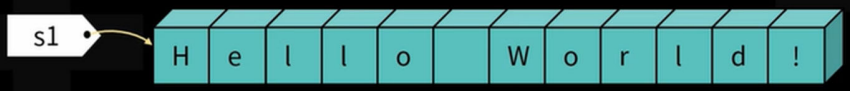
1.2 String and Variables

- So far, we have only stored integers or floats in variables, but we can also store string in variables.

```
1 s1 = 'Hello World'
2 print(s1)
```

```
Hello World
```

- Variable s1 can be seen as a tag pointing to the assigned string value.



The string data type consists of characters that make up the data.

1. String

1.3 Concatenation Operator and Repetition Operator

- We can use the addition (+) operator and multiplication (*) operator with string data, but the roles of these operators are different from numeric data, as we learned.
- In numeric data, the addition (+) operator is called the concatenation operator in String, and the multiplication (*) operator is called the repetition operator.

+	*
Concatenation operator	Repetition Operator

1. String

1.3 Concatenation Operator and Repetition Operator

- When combining two strings to form a single string, we use the '+' symbol.

```
1 s1 = "Hello"
2 s2 = "World!"
3 print(s1 + s2)
```

```
HelloWorld!
```

```
1 'I' + 'Love' + 'Python!'
```

1. String

1.3 Concatenation Operator and Repetition Operator

- When we want to repeat the same string multiple times, we use the '*' symbol and an integer. In other words, it creates the string by repeating the string n times.

```
1  'Python ' * 10
```

```
1  '*' * 50
```

```
1  str(100) * 10
```

1. String

1.3 Concatenation Operator and Repetition Operator

Focus

The * operation on string can only be used with integers. Otherwise, an error occurs.

TypeError

```
1  '*' * 3.1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 '*' * 3.1

TypeError: can't multiply sequence by non-int of type 'float'
```

One Step Further

- We learned that both string and numbers can use the addition operator, but problems occur when adding values of different data types.
- Therefore, we need to convert them to the same data type before performing the operation.

TypeError

```
1  "A" + 1
```

Adding a string and a number together will result in an error.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 "A" + 1

TypeError: can only concatenate str (not "int") to str
```

One Step Further

- We learned that both string and numbers can use the addition operator, but problems occur when adding values of different data types.
- Therefore, we need to convert them to the same data type before performing the operation.

TypeError

```
1 "10" + 10
```

'10' looks like a number, but it is a string because it is enclosed in quotes, so an error will occur.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 "10" + 10

TypeError: can only concatenate str (not "int") to str
```

1. String

1.4 String Indexing

- We learned that we can see the data type by using the type statement inside the print statement. In Python, string is of the str type.

```
1 print(type('This is a string!'))
```

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

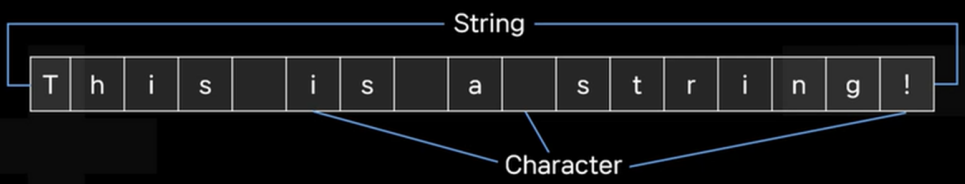
1. String

1.4 String Indexing

- Python string is a sequence of individual characters.

```
1 print(type('This is a string!'))
```

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



- Therefore, we can extract the desired part from a long string. This is called indexing.

1. String

1.4 String Indexing

- When we want to obtain a substring from a long string, we use square brackets [] and a number.
- String indexing starts from the left of the string, with the first character being 0. The 'nth' position in this sequence is called an index.

```
1 "hello"[0]
```

```
'h'
```

Line 1
- Indexing 0 retrieves the first character 'h' of 'hello'.

1. String

1.4 String Indexing

- When we want to obtain a substring from a long string, we use square brackets [] and a number.
- String indexing starts from the left of the string, with the first character being 0. The 'nth' position in this sequence is called an index.

```
1 "hello"[2]
```

```
'l'
```

Line 1
- In string indexing, using 2 as the index retrieves the third character 'l'.

1. String

1.4 String Indexing

- Python uniquely supports negative indexing, which refers to characters from the right side of the string.

```
1 "hello"[-1]
```

```
'o'
```

Line 1
- When using -1 as the index, the last character is indexed.

1. String

1.4 String Indexing

- Python uniquely supports negative indexing, which refers to characters from the right side of the string.

```
1 "hello"[-5]
```

```
'h'
```

Line 1
- When using -5 as the index, the first character is indexed.

1. String

1.4 String Indexing

- Pay attention to the indexing range.

TypeError

```
1 "hello"[5]
```

```
IndexError
Cell In[12], line 1
----> 1 "hello"[5]

IndexError: string index out of range
```

Line 1
- The string 'hello' consists of five characters.
- Therefore, the range of index is 0 to 4. If you go beyond this range, an error will be displayed, so be careful.

1. String

1.5 Multiline String

- As we learned in multiline comments, by entering three consecutive single quotes (') or double quotes (") at the beginning and end of a sentence, we can create multiline String.

```
1 text = '''the first line
2 the second line'''
3 print(text)
```

```
the first line
the second line
```

```
1 text = """the first line
2 the second line"""
3 print(text)
```

```
the first line
the second line
```

1. String

1.5 Multiline String

- We can also create multiline strings by using a backslash (\) and the letter 'n' together.

```
1 text = 'the first line\nthe second line'
2 print(text)
```

```
the first line
the second line
```

1. String

1.6 Writing Quotes within String

- If you need to use a single quote (') inside a string, enclose it in double quotes (").
  Conversely, if you need to use a double quote (") inside a string, enclose it in single quotes (').

```
1 text = '"double quotation marks"'
2 print(text)
```

```
"double quotation marks"
```

```
1 text = "'single quotation marks'"
2 print(text)
```

```
'single quotation marks'
```

1. String

1.6 Writing Quotes within String

- Another way is to use the backslash (\) and quotes together. In this case,
  you can enclose the string with either single quotes or double quotes, regardless of the type of quote within the string.

```
1 text = "\"double quotation marks\""
2 print(text)
```

```
"double quotation marks"
```

```
1 text = '\'single quotation marks\''
2 print(text)
```

```
'single quotation marks'
```

1. String

1.7 Escape sequence

- An escape sequence is a special way of representing string that have special meanings. It is created by using a backslash (\) and a specific character to represent certain special characters within a string.
  - \n: Newline character
  - \': Single quote
  - \": Double quote
  - \t: Tab character

1. String

1.7 Escape sequence

- To include a tab character, use \t.

```
1 text = 'Write a \ttab in the middle of a sentence'
2 print(text)
```

```
Write a         tab in the middle of a sentence
```

1. String

1.7 Escape sequence

- To include a tab character, use \t.

```
1 text = 'Write a \ttab in the middle of a sentence'
2 print(text)
```

```
Write a         tab in the middle of a sentence
```

Line 1

- If you want to include a tab character in the middle of a sentence, use \t at that position.

1. String

1.8 String formatting

- String formatting is a way to insert variables, values, and expressions into a string.
- By using string formatting, you can dynamically insert changing values into a string in the desired format. It uses the format statement provided by String.

```
1 value1 = 1
2 value2 = '2'
3 'first value {} and second value {}'.format(value1, value2)
```

```
'first value 1 and second value 2'
```

```
1 'Hello {}!'.format('Python')
```

```
'Hello Python!'
```

```
1 'first value {} and second value {}'.format(1, 2)
```

1. String

1.8 String formatting

```
value1 = 1
value2 = '2'
'first value {} and second value {}'.format(value1, value2)
```

placeholder

Content to be placed in placeholder

```
'first value 1 and second value 2'
```

```
'Hello {}!'.format('Python')
```

The curly braces {} inside the string are also called placeholder.

```
'Hello Python!'
```

```
'first value {} and second value {}'.format(1, 2)
```

```
'first value 1 and second value 2'
```

```
'{} + {} = {}'.format(1, 2, 1 + 2)
```

```
'1 + 2 = 3'
```

1. String

1.8 String formatting

- You can specify the order as an integer value inside the placeholder.
  If no order is specified, it defaults to 0, 1, etc. in order.

```
'I like {} and {}'.format('Python', 'Java')
```

```
'I like Python and Java'
```

```
'I like {0} and {1}'.format('Python', 'Java')
```

```
'I like Python and Java'
```

```
'I like {1} and {0}'.format('Python', 'Java')
```

```
'I like Java and Python'
```

1. String

1.8 String formatting

```
'I like {} and {}'.format('Python', 'Java')
```

```
'I like Python and Java'
```

```
'I like {0} and {1}'.format('Python', 'Java')
```

You can insert order numbers like 0, 1, etc. inside the placeholder. Python and Java will be inserted according to these order numbers.

```
'I like Python and Java'
```

```
'I like {1} and {0}'.format('Python', 'Java')
```

```
'I like Java and Python'
```

```
'I like {} and {}'.format('Python', 'Java')
'I like {0} and {1}'.format('Python', 'Java')
'I like {1} and {0}'.format('Python', 'Java')
```

Role of placeholder and index

1. String

1.8 String formatting

- Let's create various string by specifying numbers inside the placeholder.

```
'{0}, {0}, {0}! Python'.format('Hello')
```

```
'Hello, Hello, Hello! Python'
```

```
'{0}, {0}, {0}! Python'.format('Hello', 'Hi')
```

```
'Hello, Hello, Hello! Python'
```

```
'{0} {1}, {0} {1}, {0} {1}!'.format('Hello', 'Python')
```

```
'Hello Python, Hello Python, Hello Python!'
```

```
'{0} {1}, {0} {1}, {0} {1}!'.format(100, 200)
```

```
'100 200, 100 200, 100 200!'
```

1. String

1.8 String formatting

- When specifying a field width using a colon (:) inside the placeholder, you can also specify the spacing between characters, i.e., the field width. You must include a number after the colon.
- Using {:10} takes up ten spaces.

```
'{:10}'.format(123)
```

```
'       123'
```

```
'{:10}'.format('Hello')
```

```
'Hello     '
```

1. String

1.8 String formatting

'{:10}'.format(123)

'   123'

'{:10}'.format('Hello')

'Hello   '

- In the above code, a field width of :10 is specified inside the {}. You can see that even for numbers, spaces appear on the left, and for String, spaces appear on the right.
- You can specify the alignment based on whether the value is a numeric type or a string type.

1. String

1.8 String formatting

- When specifying a field width using a colon (:) inside the placeholder, you can also specify the alignment.
  - {:<10}: Field width of 10, left alignment
  - {:>10}: Field width of 10, right alignment
  - {:^10}: Field width of 10, center alignment

1. String

1.8 String formatting

- An example code that specifies the alignment and field width for numeric types is as follows.

```
1  value = 123
2  formatted_value = '{:10}'.format(value)
3  print(formatted_value)
4
5  formatted_value = '{:<10}'.format(value)
6  print(formatted_value)
7
8  formatted_value = '{:>10}'.format(value)
9  print(formatted_value)
10
11 formatted_value = '{:^10}'.format(value)
12 print(formatted_value)
```

1. String

1.8 String formatting

- An example code that specifies the alignment and field width for numeric types is as follows.

```
5  formatted_value = '{:<10}'.format(value)
6  print(formatted_value)
7
8  formatted_value = '{:>10}'.format(value)
9  print(formatted_value)
10
11 formatted_value = '{:^10}'.format(value)
12 print(formatted_value)
```

```
       123
123
       123
  123
```

1. String

1.8 String formatting

- An example code that specifies the alignment and field width for string is as follows.

```
1  value = 'Hello'
2  formatted_value = '{:10}'.format(value)
3  print(formatted_value)
4
5  formatted_value = '{:<10}'.format(value)
6  print(formatted_value)
7
8  formatted_value = '{:>10}'.format(value)
9  print(formatted_value)
10
11 formatted_value = '{:^10}'.format(value)
12 print(formatted_value)
```

```
Hello
Hello
```

```
3  print(formatted_value)
4
5  formatted_value = '{:<10}'.format(value)
6  print(formatted_value)
7
8  formatted_value = '{:>10}'.format(value)
9  print(formatted_value)
10
11 formatted_value = '{:^10}'.format(value)
12 print(formatted_value)
```

```
Hello
Hello
     Hello
  Hello
```

1. String

1.8 String formatting

```
1  value = 'Hello'
2  formatted_value = '{:10}'.format(value)
3  print(formatted_value)
4
5  formatted_value = '{:<10}'.format(value)
6  print(formatted_value)
7
8  formatted_value = '{:>10}'.format(value)
9  print(formatted_value)
10
11 formatted_value = '{:^10}'.format(value)
12 print(formatted_value)
```

Line 3

- Unlike numeric types, string is left-aligned by default, leaving space on the right.

```
Hello
Hello
     Hello
  Hello
```

1. String

1.8 String formatting

```
1  value = 'Hello'
2  formatted_value = '{:10}'.format(value)
3  print(formatted_value)
4
5  formatted_value = '{:<10}'.format(value)
6  print(formatted_value)
7
8  formatted_value = '{:>10}'.format(value)
9  print(formatted_value)
10
11 formatted_value = '{:^10}'.format(value)
12 print(formatted_value)
```

```
Hello
Hello
     Hello
  Hello
```

Line 8
- Therefore, to create space on the left for string like for numeric types, use right alignment.

1. String

1.8 String formatting

- By using d inside the placeholder, you can receive integer values, and by using f, you can receive floating-point values, including numbers with decimal points.

e.g.
- {:10d} takes up ten spaces and receives integer values.
- {:10f} takes up ten spaces and receives floating-point values.

1. String

1.8 String formatting

- {:10f} fills the empty space with 0.

```
1 '{:10d}'.format(123)
```

```
'       123'
```

```
1 '{:10f}'.format(123.5)
```

```
'123.500000'
```

```
1 '{:10f}'.format(123)
```

```
'123.000000'
```

- {:10f} can also receive integer values.

1. String

1.8 String formatting

- Since floating-point numbers include integers in mathematics, {:10f} can also receive integer values. However, {:10d} can only receive integer values.

Focus

Be careful because a ValueError may occur if you insert a floating-point value into {:10d}.

1. String

1.8 String formatting

- Since floating-point numbers include integers in mathematics, {:10f} can also receive

TypeError

```
1 '{:10d}'.format(123.5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[63], line 1
----> 1 '{:10d}'.format(123.5)

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

1. String

1.8 String formatting

- Since {:10f} fills the empty space with 0, it is meaningless to use it with alignment symbols.

```
1  value = 123.5
2  formatted_value = '{:10f}'.format(value)
3  print(formatted_value)
4
5  formatted_value = '{:<10f}'.format(value)
6  print(formatted_value)
7
8  formatted_value = '{:>10f}'.format(value)
9  print(formatted_value)
10
11 formatted_value = '{:^10f}'.format(value)
12 print(formatted_value)
```

```
123.500000
123.500000
```

```
print(formatted_value)

formatted_value = '{:<10f}'.format(value)
print(formatted_value)

formatted_value = '{:>10f}'.format(value)
print(formatted_value)

formatted_value = '{:^10f}'.format(value)
print(formatted_value)
```

```
123.500000
123.500000
123.500000
123.500000
```

1. String

1.8 String formatting

- Floating-point numbers with decimal places can be expressed in placeholder using f.
  {:10f} takes up ten spaces and receives a value as a floating-point number.
  The letter f represents a floating point.
- By adding a dot and a number in front of f, you can specify how many decimal places to display.

e.g. {:10.3f} takes up ten spaces and displays up to three decimal places.

1. String

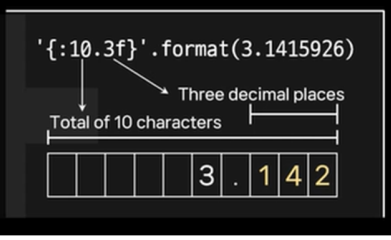
1.8 String formatting

```
'{:10.3f}'.format(3.1415926)
```

```
'     3.142'
```

```
'{:10.4f}'.format(3.1415926)
```

```
'    3.1416'
```

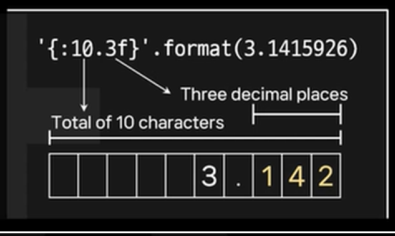


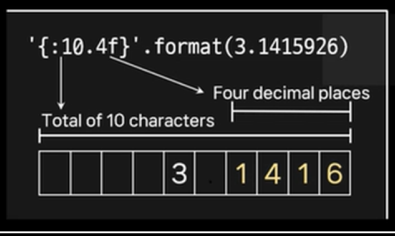
1. String

1.8 String formatting

```
1 '{:10.3f}'.format(3.1415926)
'     3.142'
```

```
1 '{:10.4f}'.format(3.1415926)
'    3.1416'
```





1. String

1.9 String Splitting

- String splitting is the process of dividing a string into multiple strings based on a delimiter.
- By using the split function provided by String, you can split a single string into one or more String. By default, whitespace is used as the delimiter for splitting the string.

1. String

1.9 String Splitting

- After the string, you use a dot (.) and then write the split statement. It will return one or more String. The returned values are assigned using the simultaneous assignment statement that you learned earlier.

```
1 'Python is beautiful'.split()
['Python', 'is', 'beautiful']
```

```
1 word1, word2, word3 = 'Python is beautiful'.split()
2 print(word1, word2, word3)
Python is beautiful
```

1. String

1.9 String Splitting

- After the string, you use a dot (.) and then write the split statement. It will return one or more String. The returned values are assigned using the simultaneous assignment statement that you learned earlier.

```
'Python is beautiful'.split()
```

```
['Python', 'is', 'beautiful']
```

```
word1, word2, word3 = 'Python is beautiful'.split()
print(word1, word2, word3)
```

```
Python is beautiful
```

- You can see that one or more string are enclosed in square brackets ([]). This is a data type called a 'list,' and you will learn more about it later.

1. String

1.9 String Splitting

- The delimiter that separates a single string into one or more individual strings is called a separator. You put the separator inside the split function. This allows you to split the string based on the separator.

```
year, month, day = '2023.5.15'.split('.')
print(year, month, day)
```

```
2023 5 15
```

- If there are dates separated by dots (.) such as year, month, and day, you can separate the string by dot units.

1. String

1.9 String Splitting

```
text1, text2 = 'Hello, World'.split(',')
print(text1, text2)
```

```
Hello  World
```

- The string was separated by a comma (,).

1. String

1.10 Checking if a String is a Numeric Character

- We learned that enclosing numerical data with quotation marks turns it into a string.
- Among them, the characters that clearly represent numbers from 0 to 9 within a string are called 'digit characters.' There is a way to check if a string is composed of 'digit characters.'

1. String

1.10 Checking if a String is a Numeric Character

- The isdigit statement checks each character of the given string and returns True or False, indicating whether it consists only of numeric characters.

```
1  string1 = '12345'
2  print(string1.isdigit())
3
4  string2 = 'Hello'
5  print(string2.isdigit())
6
7  string3 = '3.14'
8  print(string3.isdigit())
9
10 string4 = '-42'
11 print(string4.isdigit())
12
13 string5 = '0'
```

```
7  string3 = '3.14'
8  print(string3.isdigit())
9
10 string4 = '-42'
11 print(string4.isdigit())
12
13 string5 = '0'
14 print(string5.isdigit())
```

```
True
False
False
False
True
```

1. String

1.10 Checking if a String is a Numeric Character

```
1  string1 = '12345'
2  print(string1.isdigit())
3
4  string2 = 'Hello'
5  print(string2.isdigit())
6
7  string3 = '3.14'
8  print(string3.isdigit())
9
10 string4 = '-42'
11 print(string4.isdigit())
12
13 string5 = '0'
14 print(string5.isdigit())
```

```
True
False
False
False
True
```

- It returns True for cases where all characters are digits, such as '12345' and '0.'

1. String

1.10 Checking if a String is a Numeric Character

```
1  string1 = '12345'
2  print(string1.isdigit())
3
4  string2 = 'Hello'
5  print(string2.isdigit())
6
7  string3 = '3.14'
8  print(string3.isdigit())
9
10 string4 = '-42'
11 print(string4.isdigit())
12
13 string5 = '0'
14 print(string5.isdigit())
```

```
True
False
False
True
```

- It returns True for cases where all characters are digits, such as '12345' and '0.'
- It returns False for string that have a decimal point or negative sign, such as '3.14' and '-42.'

1. String

1.10 Checking if a String is a Numeric Character

- Sometimes it is convenient to use the isdigit function before converting a string to the int type. This is because if the data is not of that type and you try to convert it, an error will occur.

ValueError

```
1 int("Hello")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[7], line 1
----> 1 int("Hello")

ValueError: invalid literal for int() with base 10: 'Hello'
```

2. Writing Input Statements

2.1 Basic Usage of the input Statement

- Let's write a code that asks the user for their name.
- The input() statement is a command that receives a value from the user.

```
1 print('Please enter your name: ')
2 name = input()
3 print('Hello, ', name)
```

```
Please enter your name:
Emcast
Hello,  Emcast
```

- The variable 'name' contains the name entered by the user.

2. Writing Input Statements

2.1 Basic Usage of the input Statement

```
1 name = input('Please enter your name: ')
2 print('Hello, ', name)
```

```
Please enter your name: Emcast
Hello,  Emcast
```

- If you put a string inside the parentheses of the input statement, you can output the question without a print statement in front of the input statement.

2. Writing Input Statements

2.2 Getting a Number Input from the User

- The input statement returns a string. Therefore, if you need to perform calculations with a number entered by the user, you need to convert the data type.

Focus

By enclosing the input statement with the int function, you can receive an integer from the user.

```
1 x = int(input("Enter the first integer: "))
2 y = int(input("Enter the second integer: "))
3 s = x + y
4 print(x, "and", y, "sum up to", s, ".")
```

```
Enter the first integer: 300
Enter the second integer: 400
300 and 400 sum up to 700 .
```

2. Writing Input Statements

2.2 Getting a Number Input from the User

ValueError

```
1 x = int(input("Enter the first integer: "))
2 y = int(input("Enter the second integer: "))
3 s = x + y
4 print(x, "and", y, "sum up to", s, ".")
```

If you enter a character like 'hundred' instead of the number 300, the int function cannot convert it to an integer. Therefore, a runtime error occurs.

```
Enter the first integer: hundred

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[25], line 1
----> 1 x = int(input("Enter the first integer: "))
      2 y = int(input("Enter the second integer: "))
      3 s = x + y

ValueError: invalid literal for int() with base 10: 'hundred'
```

2. Writing Input Statements

2.2 Getting a Number Input from the User

Focus

By enclosing the input statement with the float statement,
you can receive a floating-point number with a decimal point from the user.

```
1 weight = float(input("Please enter your weight in kilograms(kg): "))
2 height = float(input("Please enter your height in meters(m): "))
3
4 bmi = (weight / (height ** 2))
5 print("Your BMI =", bmi)
```

```
Please enter your weight in kilograms(kg): 70
Please enter your height in meters(m): 1.75
Your BMI = 22.857142857142858
```

2. Writing Input Statements

2.3 Splitting an Input String from the User

- By using the split statement on the string received from the input statement, you can receive multiple values from the user at once.

```
1 num1, num2, num3 = input('Enter three integers separated by space: ').split()
2 print(num1, num2, num3)
```

```
Enter three integers separated by space: 1 2 3
1 2 3
```

2. Writing Input Statements

2.3 Splitting an Input String from the User

- By using the split statement on the string received from the input statement, you can receive multiple values from the user at once.

```
1 num1, num2, num3 = input('Enter three integers separated by space: ').split()
2 print(num1, num2, num3)
```

```
Enter three integers separated by space: 1 2 3
1 2 3
```

Input separated by spaces

2. Writing Input Statements

2.3 Splitting an Input String from the User

- By using the split statement on the string received from the input statement, you can receive multiple values from the user at once.

```
1 num1, num2, num3 = input('Enter three integers separated by comma: ').split(',')
2 print(num1, num2, num3)
```

```
Enter three integers separated by comma: 1,2,3
1 2 3
```

Input separated by commas

2. Writing Input Statements

2.3 Splitting an Input String from the User

- Split values using the split statement are string, so you need to convert them to the int type to perform numerical operations.

```
num1, num2, num3 = input('Enter three integers separated by comma: ').split(',')
num1, num2, num3 = int(num1), int(num2), int(num3)
print('{} + {} + {} = {}'.format(num1, num2, num3, num1 + num2 + num3))
```

```
Enter three integers separated by comma: 1,2,3
1 + 2 + 3 = 6
```

2. Writing Input Statements

2.4 Writing Input Statements with String Formatting

- You can apply string formatting to the input statement to output a string that corresponds to the question.

```
num = 1
text = 'name'
name = input('Q{}. Please enter your {}: '.format(num, text))
print(name)
```

```
Q1. Please enter your name: Emcast
Emcast
```

2. Writing Input Statements

2.5 Change Calculator Program

- Let's create a calculator that takes input from the user for the money they put in and the price of the item, and outputs the change and the number of coins.
- However, there are only 500 won and 100 won coins.

2. Writing Input Statements

2.5 Change Calculator Program

```
1  money = int(input("Money you put in: "))
2  price = int(input("Price of goods: "))
3
4  change = money - price
5  print("Change: ", change)
6  coin500s = change // 500
7  change = change % 500
8  coin100s = change // 100
9
10 print("The number of 500 won coins: ", coin500s)
11 print("The number of 100 won coins: ", coin100s)
```

Line 1, 2
- Receive user-defined values for money and price as int types.

```
Money you put in: 5000
Price of goods: 300
Change:  4700
The number of 500 won coins:  9
The number of 100 won coins:  2
```

2. Writing Input Statements

2.5 Change Calculator Program

```
1  money = int(input("Money you put in: "))
2  price = int(input("Price of goods: "))
3
4  change = money - price
5  print("Change: ", change)
6  coin500s = change // 500
7  change = change % 500
8  coin100s = change // 100
9
10 print("The number of 500 won coins: ", coin500s)
11 print("The number of 100 won coins: ", coin100s)
```

Line 4
- Assign the change value, which is money - price, to change.

```
Money you put in: 5000
Price of goods: 300
Change:  4700
The number of 500 won coins:  9
The number of 100 won coins:  2
```

2. Writing Input Statements

2.5 Change Calculator Program

```
1  money = int(input("Money you put in: "))
2  price = int(input("Price of goods: "))
3
4  change = money - price
5  print("Change: ", change)
6  coin500s = change // 500
7  change = change % 500
8  coin100s = change // 100
9
10 print("The number of 500 won coins: ", coin500s)
11 print("The number of 100 won coins: ", coin100s)
```

```
Money you put in: 5000
Price of goods: 300
Change:  4700
The number of 500 won coins:  9
The number of 100 won coins:  2
```

Line 6
- coin500s stores the quotient when change is divided by 500.

2. Writing Input Statements

2.5 Change Calculator Program

```
1  money = int(input("Money you put in: "))
2  price = int(input("Price of goods: "))
3
4  change = money - price
5  print("Change: ", change)
6  coin500s = change // 500
7  change = change % 500
8  coin100s = change // 100
9
10 print("The number of 500 won coins: ", coin500s)
11 print("The number of 100 won coins: ", coin100s)
```

```
Money you put in: 5000
Price of goods: 300
Change:  4700
The number of 500 won coins:  9
The number of 100 won coins:  2
```

Line 7
- Assign the remainder when change is divided by 500 to change.

2. Writing Input Statements

2.5 Change Calculator Program

```
1  money = int(input("Money you put in: "))
2  price = int(input("Price of goods: "))
3
4  change = money - price
5  print("Change: ", change)
6  coin500s = change // 500
7  change = change % 500
8  coin100s = change // 100
9
10 print("The number of 500 won coins: ", coin500s)
11 print("The number of 100 won coins: ", coin100s)
```

Line 8
- coin100s stores the quotient when change is divided by 100.

```
Money you put in: 5000
Price of goods: 300
Change:  4700
The number of 500 won coins:  9
The number of 100 won coins:  2
```

One Step Further

▶ Incorrect use of expressions leads to errors.

ZeroDivisionError

```
1  num1 = 3
2  num2 = 0
3  print(num1 / num2)
```

When using mathematical operators, you should double-check whether the expression is correct.

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[14], line 3
      1 num1 = 3
      2 num2 = 0
----> 3 print(num1 / num2)

ZeroDivisionError: division by zero
```

One Step Further

▶ Incorrect use of expressions leads to errors.

ZeroDivisionError

```
1 num1 = 3
2 num2 = 0
3 print(num1 / num2)
```

An error occurs when dividing a numeric value by 0.
Make sure to read the error message carefully.

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[14], line 3
      1 num1 = 3
      2 num2 = 0
----> 3 print(num1 / num2)

ZeroDivisionError: division by zero
```

3. String and Output

3.1 The Print Statement and Commas

- When using a comma (,) in the print statement, you can output string and numerical values together. The comma (,) automatically inserts a space for output.

```
1 print('I am', 23, 'years old!')
2 print('I', 'really ' * 3, 'love', 'Python!')
```

```
I am 23 years old!
I really really really  love Python!
```

3. String and Output

3.2 The Print Statement and the End Parameter

- After executing the print statement, a line break occurs. However, if you specify a string using the 'end' keyword at the end of the print statement, the string is printed immediately after the output instead of a line break.

```
1 print('Hello', end = '')
2 print('World!')
```

```
HelloWorld!
```

```
1 print(1, end = '')
2 print(2)
```

```
12
```

- An empty string ('') is printed without any spaces.

3. String and Output

3.2 The Print Statement and the End Parameter

```
1 print('Hello', end = ',')
2 print('World!')
```

```
Hello,World!
```

- ‘,’ prints a comma.

```
1 print("My name is", end = " ")
2 print('David')
```

```
My name is David
```

- ‘ ’ prints a space.

3. String and Output

3.2 The Print Statement and the End Parameter

```
1 print("My name is", end = " : ")
2 print('David')
```

```
My name is : David
```

- ‘:’ prints a colon.

3. String and Output

3.3 Print Statement and Separator

- By specifying a separator using the 'sep' keyword at the end of the print statement, a separator is printed between the values to be output.

```
1 print('Hello', 'World!', sep = '')
```

```
HelloWorld!
```

- This code prints 'Hello' followed immediately by 'World!' without a space.

```
1 print(1, 2, sep = '')
```

```
12
```

- This code prints '1' followed immediately by '2' without a space.

3. String and Output

3.3 Print Statement and Separator

```
1 print('Hello', 'World!', sep = ',')
```

```
Hello,World!
```

- This code prints a comma (,) between two strings.

```
1 print('s', 'e', 'p', sep = '-')
```

```
s-e-p
```

- This code prints a dash (-) between three strings.

3. String and Output

3.4 Output Using String Formatting

- Let's create a code that summarizes everything we have learned so far.

e.g. Receive the user's name, age, and occupation as input, store them in the variables name, age, and job, and use the format statement of String to output them.

```
1 name = input('Enter your name : ')
2 age = input('Enter your age : ')
3 job = input('Enter your job : ')
4
5 print('Your name is {}, and {} years old, job is {}.'.format(name, age, job))
```

```
Enter your name : David
Enter your age : 23
Enter your job : Programmer
Your name is David, and 23 years old, job is Programmer.
```

4. Errors and Debugging

4.1 Syntax Error

- Programming languages are executed based on strict rules. Syntax errors occur when a command violates the prescribed grammar when coding in Python.
- To prevent syntax errors, you need to learn the correct Python syntax and code according to the prescribed rules.

4. Errors and Debugging

4.1 Syntax Error

- To output the string 'Hello', you need to use quotation marks on both sides of the string. However, the following code did not include quotation marks after the string, resulting in a syntax error.

```
1 print('Hello)
```

```
Cell In[39], line 1
  print('Hello)
        ^
SyntaxError: unterminated string literal (detected at line 1)
```

4. Errors and Debugging

4.2 Runtime Error

- Unlike syntax errors, even sentences written in accordance with Python syntax can cause errors during execution. These errors are called runtime errors.
- To prevent runtime errors, as a developer, you need to consider that users can enter incorrect data and send user-friendly and helpful input request messages.
- Additionally, since exceptional cases can occur in all user inputs and code, it is necessary to perform many tests for various situations.

4. Errors and Debugging

4.2 Runtime Error

- In the following code, a runtime error occurred because the user entered the string 'two' instead of the integer 2 during program execution.

```
1 num = int(input('Input an integer: '))
```

```
Input an integer: two

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[40], line 1
----> 1 num = int(input('Input an integer: '))

ValueError: invalid literal for int() with base 10: 'two'
```

5. Built-in Function

5.1 Functions Represented as Black Boxes

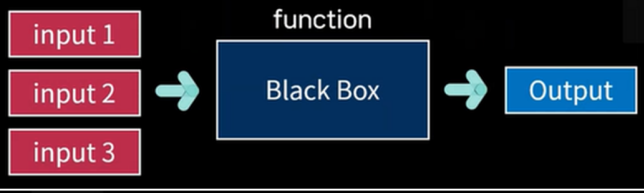
- The print and input statements we have used so far are called function.
- In programming, functions are often compared to black boxes.
  When calling a function, the user only needs to provide the correct input values and use the output (result) without worrying about how the function works.

5. Built-in Function

5.1 Functions Represented as Black Boxes

- There is a term used to refer to the input values provided when calling a function.
  Let's take a closer look at some key terms related to functions.
- Parameter: Variables defined inside the parentheses, which are variable that receive actual values when the function is called.

e.g. In the case of def foo(m, n):, 'm' and 'n' are parameters.



5. Built-in Function

5.1 Functions Represented as Black Boxes

- There is a term used to refer to the input values provided when calling a function. Let's take a closer look at some key terms related to functions.
- Argument: The actual value passed when the function is called, also referred to as argument.

e.g. In the case of foo(3, 4), '3' and '4' are arguments.

5. Built-in Function

5.2 Python's Built-in Functions

- Functions that are implemented by default in Python and provided are called Python's built-in function.
- The print function and input function are also part of these functions.
- Python provides over 100 built-in functions.

5. Built-in Function

5.2 Python's Built-in Functions

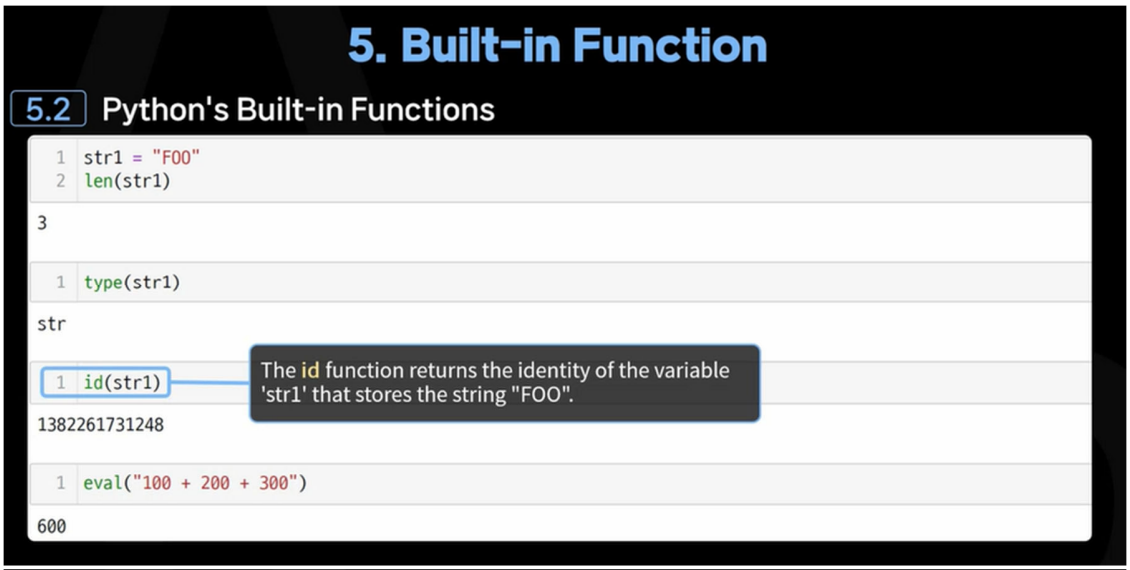
Python's built-in functions				
abs	dict	help	min	setattr
all	dir	hex	next	slice
any	divmod	id	object	sorted
ascii	enumerate	input	oct	staticmethod
bin	eval	int	open	str
bool	exec	isinstance	ord	sum
bytearray	filter	issubclass	pow	super
bytes	float	iter	print	tuple
callable	format	len	property	type

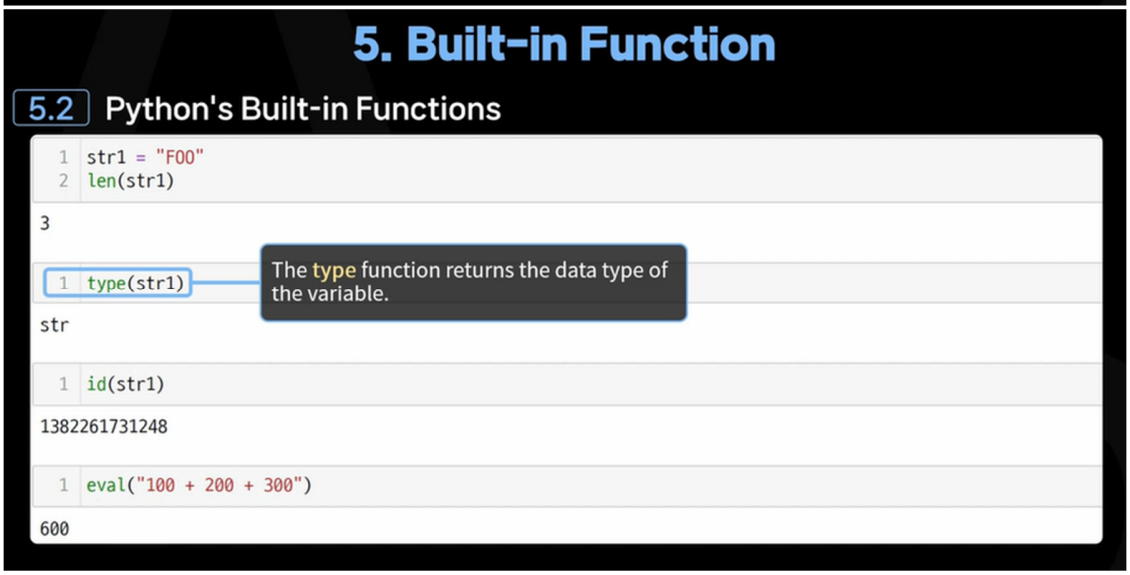
5. Built-in Function

5.2 Python's Built-in Functions

Python's built-in functions				
chr	frozenset	list	range	vars
classmethod	getattr	locals	repr	zip
compile	globals	map	reversed	_import_
complex	hasattr	max	round	
delattr	hash	memoryview	set	

- If you want to know more about the list of built-in functions, refer to https://docs.python.org/3/library/functions.html.


5. Built-in Function
5.2 Python's Built-in Functions
1 str1 = "FOO"
2 len(str1)
3
1 type(str1)
str
1 id(str1)
The id function returns the identity of the variable 'str1' that stores the string "FOO".
1382261731248
1 eval("100 + 200 + 300")
600


5. Built-in Function
5.2 Python's Built-in Functions
1 str1 = "FOO"
2 len(str1)
3
1 type(str1)
The type function returns the data type of the variable.
str
1 id(str1)
1382261731248
1 eval("100 + 200 + 300")
600

5. Built-in Function

5.2 Python's Built-in Functions

```
1 str1 = "FOO"
2 len(str1)
```

The len function returns the length of the string stored in the variable 'str1'.

```
3
```

```
1 type(str1)
```

```
str
```

```
1 id(str1)
```

```
1382261731248
```

```
1 eval("100 + 200 + 300")
```

```
600
```

5. Built-in Function

5.2 Python's Built-in Functions

```
1 str1 = "FOO"
2 len(str1)
```

```
3
```

```
1 type(str1)
```

```
str
```

```
1 id(str1)
```

```
1382261731248
```

```
1 eval("100 + 200 + 300")
```

The eval function takes a string, evaluates the contents of the string as an expression, and returns the evaluated value.

```
600
```

5. Built-in Function

5.2 Python's Built-in Functions

```
1 abs(-100)
```

The abs function takes the integer -100 as input and returns its absolute value, 100.

```
100
```

```
1 min(200, 100, 300, 400)
```

```
100
```

```
1 max(200, 100, 300, 400)
```

```
400
```

5. Built-in Function

5.2 Python's Built-in Functions

```
1 abs(-100)
```

```
100
```

```
1 min(200, 100, 300, 400)
```

The min function returns the smallest value among the integers 200, 100, 300, and 400.

```
100
```

```
1 max(200, 100, 300, 400)
```

```
400
```

5. Built-in Function

5.2 Python's Built-in Functions

```
1 abs(-100)
```

```
100
```

```
1 min(200, 100, 300, 400)
```

```
100
```

```
1 max(200, 100, 300, 400)
```

The max function returns the largest value among the integers 200, 100, 300, and 400.

```
400
```

5. Built-in Function

5.2 Python's Built-in Functions

- The following code sorts a string in ascending or descending order.

```
1 sorted('EABFD')
```

```
['A', 'B', 'D', 'E', 'F']
```

  - The sorted function takes a string and returns the characters that make up the string sorted in alphabetical order.
    - You can see one or more String inside square brackets ([]). This is a data type called 'list', and we will learn more about it later.

5. Built-in Function

5.2 Python's Built-in Functions

```
1 sorted('EABFD', reverse = True)
```

```
['F', 'E', 'D', 'B', 'A']
```

  - If you set reverse = True, it will be sorted in descending order.

5. Built-in Function

5.3 Difference between Function and Method

- The format statement and the split statement we learned for String are called method.
- A method is like black boxes that take inputs, perform a specific action, and return results, just like a function. However, there are differences between the two.
- Unlike functions, methods cannot be called independently because they belong to specific objects. They need to be called using a dot (.) after the object's name.
  - Therefore, we have been using the dot notation to call the format method and the split method for String.
  - We learned that data types like String are independent objects designed to perform specific tasks. Objects and methods will be discussed in more detail in future classes.

6. Built-in Modules

6.1 Modules and the Import Statement

- Module
  - A file that collects Python functions, variables, etc.
  - Python has many modules developed by numerous developers.

6. Built-in Modules

6.1 Modules and the Import Statement

- The keyword used to import modules is "import".
  - When importing a created module, we write the module name after "import".

```
import module_name1 [, module_name2, ...]
```

  - Alternatively, when using it, we can write the module name followed by a dot (.) and then write the components of the module.

```
import [module name].[class name].[method name]
```

6. Built-in Modules

6.1 Modules and the Import Statement

- By using the "as" syntax, we can use an alias for the module name.

```
import [module name] as [module alias]
```

- Using the "as" syntax, we can shorten the module name datetime to a simpler form, such as dt.

```
1 import datetime as dt
2 start_time = dt.datetime.now()
3 start_time.replace(month = 12, day = 25)
```

```
datetime.datetime(2023, 12, 25, 15, 59, 13, 394983)
```

One Step Further

- Import-as statement:
  - ➡ A way to assign a nickname to a long module name, allowing it to be called more easily when using classes or methods within the module.
  - ➡ When using classes or methods within the module, we connect them with dots. If the module name is too long, it can be cumbersome to keep typing it repeatedly.
  - ➡ In this case, we can use "as" to assign a new name to the module.

One Step Further

- Import-as statement:
  - ➡ Commonly used short names for modules include m for math, dt for datetime, rd for random, and t for turtle.

e.g. 1 import datetime as dt

e.g. 2 import random as rd

e.g. 3 import math as m

e.g. 4 import turtle as t

6. Built-in Modules

6.1 Modules and the Import Statement

- The from-import statement is a way to import only what is needed from a module.

```
"from [module name] import [class name].[method name]"
```

↳ Using the from statement allows us to omit the "module name" that was required in the import statement. The square brackets ([ ]) indicate that it is optional.

6. Built-in Modules

6.1 Modules and the Import Statement

- The datetime module only includes the datetime class.

```
1 from datetime import datetime
2 start_time = datetime.now()
3 start_time.replace(month = 12, day = 25)
```

```
datetime.datetime(2023, 12, 25, 21, 3, 1, 878961)
```

6. Built-in Modules

6.1 Modules and the Import Statement

- "from [module name] import *" means to import all variables, functions, and classes from the module.
  ↳ "*" means "everything".
- The code below imports all classes from the datetime module and uses the today method of the date class to print today's date.

```
1 from datetime import *
2 today = date.today()
3 today
```

```
datetime.date(2023, 5, 17)
```

6. Built-in Modules

6.1 Modules and the Import Statement

- Let's learn about the modules necessary for learning Python.
  - Modules provided with Python installation are called the Python Standard Library or **built-in module**.
  - They provide over 100 built-in modules, including modules for string and text processing, binary data processing, date, time, array data types, numerical operations and math functions, file and directory access, accessing databases in Unix systems, data compression, and graphics modules.

6. Built-in Modules

6.2 The Datetime Module for Receiving and Manipulating Dates and Times

- datetime module:
  - A module that provides functionality for working with and manipulating dates and times.
- In datetime.datetime.now, the first datetime is the module name, and the second datetime is the class name. now is a method.
  - We learned that a class is an independent object designed to perform a specific task. It will be discussed in more detail in future classes.

```
1 import datetime
2 datetime.datetime.now()
```

```
datetime.datetime(2023, 5, 17, 16, 6, 42, 965030)
```

Line 2
- The datetime.now method returns the current year, month, day, hour, minute, second, and microsecond.

6. Built-in Modules

6.2 The Datetime Module for Receiving and Manipulating Dates and Times

- The replace method is used when we want to change the date or time value.

```
1 start_time = datetime.datetime.now()
2 start_time.replace(month = 12, day = 25)
```

```
datetime.datetime(2023, 12, 25, 16, 8, 13, 259168)
```

6. Built-in Modules

6.2 The Datetime Module for Receiving and Manipulating Dates and Times

- The replace method is used when we want to change the date or time value.

```
1 start_time = datetime.datetime.now()
2 start_time.replace(month = 12, day = 25)
```

```
datetime.datetime(2023, 12, 25, 16, 8, 13, 259168)
```

Line 1
- start_time is assigned the current date.

6. Built-in Modules

6.2 The Datetime Module for Receiving and Manipulating Dates and Times

- The replace method is used when we want to change the date or time value.

```
1 start_time = datetime.datetime.now()
2 start_time.replace(month = 12, day = 25)
```

```
datetime.datetime(2023, 12, 25, 16, 8, 13, 259168)
```

Line 2
- The replace method is used to change the month and day of the current date to 12 and 25, respectively.

6. Built-in Modules

6.2 The Datetime Module for Receiving and Manipulating Dates and Times

- The today method of the date class prints the current year, month, and day.

```
1 today = datetime.date.today()
2 print(today)
```

```
2023-05-17
```

```
1 today
```

```
datetime.date(2023, 5, 17)
```

```
1 today.year
```

```
2023
```

```
1 today.month
```

```
5
```

```
1 today.day
```

```
17
```

6. Built-in Modules

6.2 The Datetime Module for Receiving and Manipulating Dates and Times

- Let's use the dir function to print the functions and variables that the module has.
  - MAXYEAR has a value of 9999, which is the maximum representable year for a datetime object.
  - MINYEAR has a value of 1.
  - Classes such as date, datetime, datetime_CAPI, time, timedelta, timezone, tzinfo have convenient features for working with dates, times, time zones, and time zone information.

```
1 print(dir(datetime))
```

```
['MAXYEAR', 'MINYEAR', 'UTC', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'date', 'datetime', 'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']
```

6. Built-in Modules

6.3 The Time Module

- The time module provides information related to time.
  - It provides information related to time.
  - The epoch is the starting time for Unix systems, which is January 1, 1970, 0:00:00 Coordinated Universal Time (UTC).
  - The time system commonly used in Unix operating systems is referred to as epoch time or Unix time.

6. Built-in Modules

6.3 The Time Module

- The following code indicates the elapsed time in seconds since the epoch time (January 1, 1970, 0:00:00).

```
1 import time
2 seconds = time.time()
3 print('Time after epoch = ', seconds)
```

```
Time after epoch =  1684307696.6218932
```

6. Built-in Modules

6.3 The Time Module

- Let's use the localtime method of the time module.
  - The localtime method takes an epoch time value and returns the local time as a date and time format.

```
1 import time
2 unix_timestamp = time.time()
3 local_time = time.localtime(unix_timestamp)
4 local_time
```

```
time.struct_time(tm_year=2023, tm_mon=5, tm_mday=17, tm_hour=16, tm_min=18, tm_sec=27,
tm_wday=2, tm_yday=137, tm_isdst=0)
```

6. Built-in Modules

6.3 The Time Module

- The strftime method allows us to format the time obtained from localtime according to the desired date/time format.

```
import time
unix_timestamp = time.time()
local_time = time.localtime(unix_timestamp)
print(time.strftime('%Y-%m-%d %H-%M-%S', local_time))
```

```
2023-05-17 16-19-31
```

- %Y represents the year, %m represents the month, and %d represents the day. '%Y-%m-%d' represents the 'year-month-day' format.
- '%H:%M:%S' represents the hour (24-hour clock), minute (00 to 59), and second (00 to 59), respectively.

6. Built-in Modules

6.4 The Math Module

- The math module allows us to use mathematical functions.
- math module
  - A module that contains mathematical functions.
  - It defines constants such as pi (π) and the natural constant e.
  - It includes mathematical functions like sin, cos, tan, log, pow, ceil, floor, trunc, fabs, and copysign(x, y).

6. Built-in Modules

6.4 The Math Module

- Let's use the dir function to print the functions and variables that the module has.

```
import math
print(dir(math))
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asi
n', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign', 'cos', 'cos
h', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1', 'fabs', 'factoria
l', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfi
nite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainde
r', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

6. Built-in Modules

6.4 The Math Module

- Let's explore the functions in the math module.

```
import math as m
m.pow(3, 3)
```

```
27.0
```

```
m.fabs(-99)
```

```
99.0
```

```
m.ceil(2.1)
```

```
3
```

```
m.ceil(-2.1)
```

```
m.ceil(-2.1)
```

```
-2
```

```
m.floor(2.1)
```

```
2
```

```
m.log(2.71828)
```

```
0.999999327347282
```

```
m.log(100, 10)
```

```
2.0
```

6.4 The Math Module

- Mathematical functions in the math module, their descriptions, and usage examples.

Function	Description	Example
fabs(x)	Returns the absolute value of a real number x.	fabs(-2) → 2.0
ceil(x)	Returns x rounded up to the nearest integer.	ceil(2.1) → 3, ceil(-2.1) → -2
floor(x)	Returns x rounded down to the nearest integer.	floor(2.1) → 2, ceil(-2.1) → -3
exp(x)	Returns the value of e^x.	exp(1) → 2.71828
log(x)	Returns the natural logarithm of x.	log(2.71828) → 1.0
log(x, base)	Returns the logarithm of x to the base specified by base.	log(100, 10) → 2.0

6.4 The Math Module

- Mathematical functions in the math module, their descriptions, and usage examples.

Function	Description	Example
sqrt(x)	Returns the square root of x.	sqrt(4.0) → 2.0
sin(x)	Returns the sine (sin) value of x, where x is in radians.	sin(3.14159/2) → 1, sin(3.14159) → 0
asin(x)	Returns the arcsine (asin) value of x.	asin(1.0) → 1.57, asin(0.5) → 0.523599
cos(x)	Returns the cosine (cos) value of x, where x is in radians.	cos(3.14159/2) → 0, cos(3.14159) → -1

6. Built-in Modules

6.4 The Math Module

- Mathematical functions in the math module, their descriptions, and usage examples.

Function	Description	Example
acos(x)	Returns the arccosine value of x.	acos(1.0) → 0, acos(0.5) → 1.0472
tan(x)	Returns the tangent (tan) value of x, where x is in radians.	tan(3.14159/4) → 1, tan(0.0) → 0
degrees(x)	Converts angle x from radians to degrees.	degrees(1.57) → 90
radians(x)	Converts angle x from degrees to radians.	radians(90) → 1.57

6. Built-in Modules

6.4 The Math Module

- Let's find out why the value of sin(90) in the following code is not 1.

```
1 import math as m
2
3 m.sin(0.0)
```

```
0.0
```

```
1 m.sin(90.0)
```

```
0.8939966636005579
```

The trigonometric functions in Python's math module use radian angles as argument values.

```
1 m.sin(m.pi / 2)
```

```
1.0
```

6. Built-in Modules

6.4 The Math Module

- To use the angle 90 degrees in the math module, it needs to be converted to radians using the expression math.pi/2.0.

```
1 import math as m
2
3 m.sin(0.0)
```

```
0.0
```

```
1 m.sin(90.0)
```

```
0.8939966636005579
```

```
1 m.sin(m.pi / 2)
```

```
1.0
```

6. Built-in Modules

6.5 The Random Module

- The random module includes functions to generate random numbers, shuffle elements in a list, and select elements randomly.
- For convenience, the random module is commonly used with the alias "rd" as follows.

```
import ramdom as rd
```

6. Built-in Modules

6.5 The Random Module

- The functions in the random module and their descriptions are as follows.

Function	Description
random()	Generates a float between 0 and 1 (excluding 1).
randrange(a, b, step)	Returns a random integer N from the range a <= N < b with a step value of step.
randint(a, b)	Returns a random integer N from the range a <= N <= b.
shuffle(seq)	Shuffles the elements in the given seq list randomly.
choice(seq)	Selects a random element from the seq sequence.
sample()	Selects a specified number of elements randomly.

6. Built-in Modules

6.5 The Random Module

```
1  import random as rd
2
3  print(rd.random())
4  print(rd.random())
5  print(rd.randrange(1, 7))
6  print(rd.randrange(0, 10, 2))
7  print(rd.randint(1, 10))
```

```
0.3305135018034717
0.8560420854525972
1
0
1
```

6. Built-in Modules

6.5 The Random Module

```
1  import random as rd
2
3  print(rd.random())
4  print(rd.random())
5  print(rd.randrange(1, 7))
6  print(rd.randrange(0, 10, 2))
7  print(rd.randint(1, 10))
```

```
0.3305135018034717
0.8560420854525972
1
0
1
```

- random(): Returns a float between 0 (inclusive) and 1 (exclusive). It returns a different float each time it is used.

6. Built-in Modules

6.5 The Random Module

```
1  import random as rd
2
3  print(rd.random())
4  print(rd.random())
5  print(rd.randrange(1, 7))
6  print(rd.randrange(0, 10, 2))
7  print(rd.randint(1, 10))
```

```
0.3305135018034717
0.8560420854525972
1
0
1
```

- randrange(1, 7): Returns an integer from 1 (inclusive) to 7 (exclusive).

6. Built-in Modules

6.5 The Random Module

```
1 import random as rd
2
3 print(rd.random())
4 print(rd.random())
5 print(rd.randrange(1, 7))
6 print(rd.randrange(0, 10, 2))
7 print(rd.randint(1, 10))
```

```
0.3305135018034717
0.8560420854525972
1
0
1
```

- randrange(0, 10, 2): Returns an integer that is a multiple of 2 from 0 (inclusive) to 10 (exclusive).

6. Built-in Modules

6.5 The Random Module

```
1 import random as rd
2
3 print(rd.random())
4 print(rd.random())
5 print(rd.randrange(1, 7))
6 print(rd.randrange(0, 10, 2))
7 print(rd.randint(1, 10))
```

```
0.3305135018034717
0.8560420854525972
1
0
1
```

- randint(1, 10): Returns a random integer from 1 (inclusive) to 10 (inclusive).

6. Built-in Modules

6.5 The Random Module

- shuffle(): Returns the sequence with its elements shuffled in a different order each time.

```
1 numlist = [10, 20, 30, 40, 50]
2 rd.shuffle(numlist)
3 print(numlist)
```

```
[40, 30, 50, 10, 20]
```

```
1 a = list(range(1, 11))
2 rd.shuffle(a)
3 print(a)
```

```
[5, 3, 7, 2, 10, 9, 6, 4, 1, 8]
```

6. Built-in Modules

6.5 The Random Module

- choice(): Extracts a random element from the given sequence.

```
1 print(rd.choice(numlist))
```

```
50
```

- sample(): Returns elements randomly. The number of elements to be returned can be specified as an argument.

```
1 print(rd.sample(numlist, 3))
```

```
[50, 30, 20]
```

6. Built-in Modules

6.5 The Random Module

- Print 3 random integers between 1 and 10 using the sample function.

```
1 import random as rd
2
3 a = list(range(1, 11))
4 b = rd.sample(a, 3)
5 print('Random integer from 1 to 10 :', b)
```

```
Random integer from 1 to 10 : [8, 5, 3]
```

6. Built-in Modules

6.5 The Random Module

- Print 3 random integers between 1 and 10 using the sample function.

```
1  import random as rd
2
3  a = list(range(1, 11))
4  b = rd.sample(a, 3)
5  print('Random integer from 1 to 10 :', b)
```

```
Random integer from 1 to 10 : [8, 5, 3]
```

Line 3

- Assigns a list containing elements from 1 to 10 to variable a.

6. Built-in Modules

6.5 The Random Module

- Print 3 random integers between 1 and 10 using the sample function.

```
1  import random as rd
2
3  a = list(range(1, 11))
4  b = rd.sample(a, 3)
5  print('Random integer from 1 to 10 :', b)
```

```
Random integer from 1 to 10 : [8, 5, 3]
```

Line 4

- Uses the sample function to randomly sample 3 integers and assigns them to variable b.