CSM Python Book

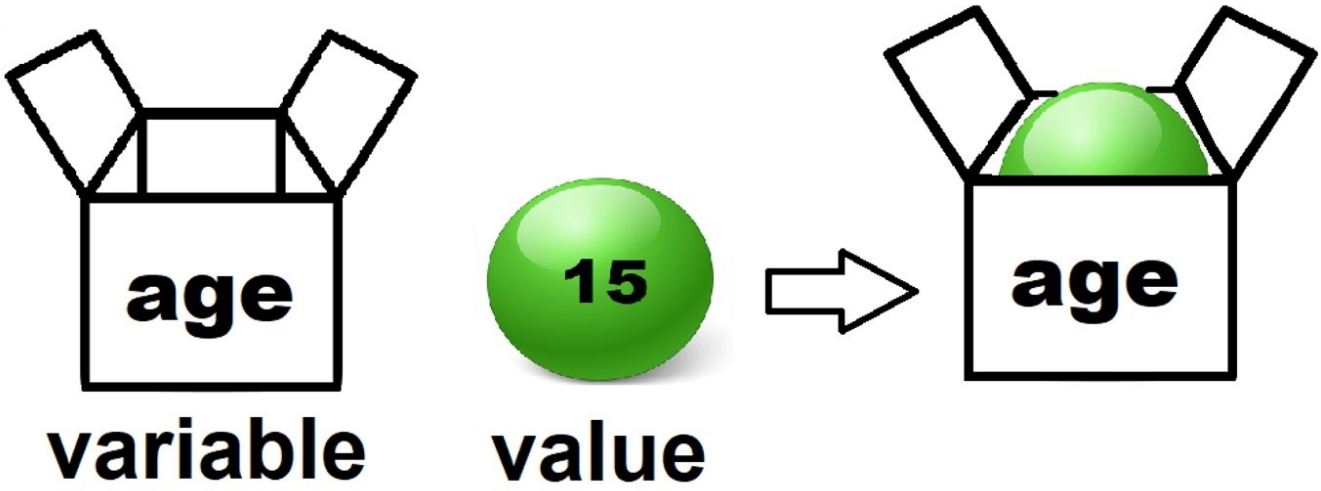
1. Chapter 1

2. Chapter 2

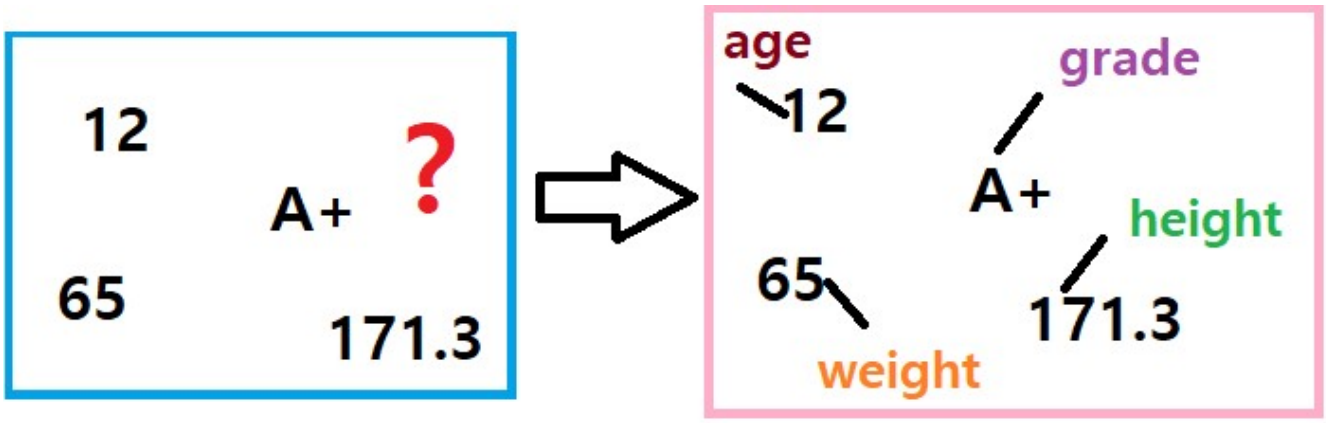
2.1. Variable

Variables are a very important and powerful tool that programmers use to store and manage data. The definition of variable is **something that can change or take different values**. For example, let's say an experiment was done on a group of teenagers of age 15. The names of each test subject will be different, so each subject's ***name*** can be treated as a variable. On the other hand, every subject's age are the same (15), therefore, ***age*** cannot be treated as a variable, but rather, a *constant*.

In programing, a variable is like a box that can store different values, such as numbers and words, inside.



Suppose there is a number 15. It can refer to anything, like someone's age or someone's quiz score. To clarify what the value means, the box, or the variable, is given a name. In the example above, the variable is named *age* to signify that 15 refers to someone's age.



Today, there exists tons of different data, which would be very hard to distinguish and organize without meaningful specifiers. Therefore, we store them inside variables with relevant names.

2.1.1. Basic Data Types in Python

It was mentioned that we can store different types of data in variables. But what are the different data types that exist in Python? The three basic types are numbers, strings, and boolean values.

1. Number

Numbers are a familiar concept of data to all of us. We use it daily, whether it is to pay for an item at a supermarket or to do our math homework. In Python, numbers are split into two parts: integers and floating point numbers.

Integers, or *int* for short, are numbers that do not have decimal points.

Examples of integers

- `1`
- `10`
- `253`
- `2,147,483,648`

Floating point numbers, or *float* for short, are numbers with decimal points.

Examples of floats

- `1.0`
- `3.14159`
- `171.3`
- `1713e-1`
- `1.713E2`

NOTE	`e` or `E` means power of 10. `1713e-1` means `1713 * 10`$^{-1}$ and `1.713E2` means `1.713 * 10`^{2}. We can see that both evaluates to `171.3`.

We can use the built-in `type(x)` function to check the data type of `x`. Try the following examples on Python shell and check the output.

`type()` *Example*

```
>>> type(1)
<class 'int'>
>>> type(2_147_483_648)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type(1713e-1)
<class 'float'>
```

NOTE	Line 3 says `2_147_483_648` instead of `2,147,483,648`. This is because commas have a special role in Python. We use underscores (_) in Python to increase readability of numbers instead of commas.

2. String

Strings refers to any sequence of characters - a 'string' of characters. To express something as text, we would use strings to do that. In Python, in order to specify that something is a string, we surround the sequence of characters in single quotation marks.

Examples of strings in Python

- `'strings'`
- `'Hello World'`
- `'I am 15 years old'`
- `'15'`

Like the last example, if numbers are surrounded by single quotation marks, Python will treat them as words, or strings, instead of a number.

Let's use the `type(x)` function again to check the data type of `x`.

`type()` *Example*

```
>>> type('Hello World')
<class 'str'>
>>> type('I am 15 years old')
<class 'str'>
>>> type('15')
<class 'str'>
```

3. Boolean

Boolean values traditionally take two values only, 0 and 1. In computer programming, Booleans are used to express two values, *True* and *False*.

Two Boolean values

- `False` (`0`)
- `True` (`1`)

In certain cases, Python will treat `True`/`False` as `1`/`0` and vice versa.

Let's use the `type(x)` function again to check the data type of `x`.

`type()` *Example*

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

As of now, it may not be clear how to use this data type. Later, we will use this data type to evaluate conditional expressions (e.g. if-else statements).

2.1.2. Variable Assignments

To effectively use variables, we must *assign* values to them. In this *assignment* step, we create a new variable with an appropriate name and give them a value.

assignment_ex.py

```
1    age = 15
2    pi = 3.14159
3    name = 'John Doe'
4    is_same = True
5    print(age, pi, name, is_same)
```

Output

```
15 3.14159 John Doe True
```

The above example showcases the variable assignments of different data types. Line 1 assigns the integer value `15` to the variable named `age`. Line 2 assigns the floating point number approximate of π to the variable `pi`. Line 3 assigns the string value `'John Doe'` to the variable `name`. Line 4 assigns the boolean value `True` to the variable `is_name`. As this example shows, with meaningful variable names, others can see what the values are referring to.

The `print()` statement in line 5 is used to display the values of each variable on the computer screen for us to see. The Output section shows how each data type looks like when printed. As shown, if the values/variables are listed by commas, they are separated by one whitespace and printed. The `print()` statement will be covered in more depth under Section 2.3, “Printing”.

Exercise 1.2-1: Variable Assignment

```
Evaluate the output of the following Python program.

1    num_int = 123
2    print('Data type of', num_int, 'is:', type(num_int))
3
4    num_float = 1618e-3
5    print('Data type of', num_float, 'is:', type(num_float))
6
7    str1 = 'Python is Fun'
8    print('The variable str1 is storing the following value:', str1)
```

Answer

Output

```
Data type of 123 is: <class 'int'>
Data type of 1.618 is: <class 'float'>
The variable str1 is storing the following value: Python is Fun
```

2.1.2.1. Multiple Assignment

We have showed that we can use 4 lines to create 4 variables, but can we do it in less? The answer is yes.

Examples of multiple assignment

```
1    a, b = 1, 2
2    age, pi, name = 15, 3.14159, 'John Doe'
```

Line 1 assigns the integer value `1` to the variable `a` and the integer value `2` to the variable `b`. The number of variables and values to assign must be equal, unless there is only one variable. But as line 2 shows, the types of values to assign may be different.

Example of multiple assignment with one variable

```
1    a = 1, 2, 3
```

If multiple values are assigned to one variable, that variable will be interpreted as a *tuple* by Python. A tuple is another data type supported by Python, not covered currently.

Examples of invalid multiple assignments

```
1    a, b = 1, 2, 3
2    a, b, c = 1, 2
```

Running the above lines of codes will cause Python to throw an error.

Exercise 1.2-2: Multiple Assignment

```
Write a Python program to assign the values 'John Doe', 175, 68, True to the variables
name, height, weight, has_siblings in one line. Check your work using the print()
statement.

Values: 'John Doe', 175, 68, True
Variables: name, height, weight, has_siblings
```

Sample Program

ex1_2_2.py

```
1    name, height, weight, has_siblings = 'John Doe', 175, 68, True
2    print(name, height, weight, has_siblings)
```

2.1.2.2. Equal Sign

Another point to note is that the equal sign (=) has a different meaning in programming than in mathematics.

In mathematics, equal sign signifies that the values on both sides are equal. `x = y` means that the value of variable `x` is the same as the value of variable `y`.

On the other hand, in computer programming, equal sign signifies that we are storing the value on the right side of the `=` sign to the variable written on the left side of the `=` sign. Therefore, `x = y` in Python means that the value of `y` is stored in variable `x`.

```
1    y = 10
2    x = y
```

In the above snippet of code, since the value of `y` is 10, line 2 is storing the integer value 10 to the variable `x`.

2.1.2.3. Naming Conventions of Variables

`Capitalized`, `PascalCase`, `camelCase` are all valid variable names. However, to standardize variable names, Python has a set of rules when naming variables.

1. Not allowed

i. Cannot contain any characters other than alphabets, numbers, and underscores

ii. Cannot start with a number

iii. Cannot be a word reserved by Python (called a keyword)

Below shows all 35 keywords reserved by Python:

`and`	`del`	`from`	`None`	`True`
`as`	`elif`	`global`	`nonlocal`	`try`
`assert`	`else`	`if`	`not`	`while`
`break`	`except`	`import`	`or`	`with`
`class`	`False`	`in`	`pass`	`yield`
`continue`	`finally`	`is`	`raise`	`async`
`def`	`for`	`lambda`	`return`	`await`

Exercise 1.2-3: Valid Variable Names

```
Are the variable names listed below valid (Yes/No)?

Q1. valid_variable
Q2. name1_
Q3. 1place
Q4. safe_password@1^
Q5. class
Q6. elif_if
```

Answer

Question	Answer	Explanation
Q1	Yes	1. Only contains letters, numbers, and underscores 2. Does not start with a number 3. Is not a keyword
Q2	Yes	1. Only contains letters, numbers, and underscores 2. Does not start with a number 3. Is not a keyword
Q3	No	Starts with a number
Q4	No	Contains characters other than letters, numbers, and underscores
Q5	No	`class` is a keyword reserved by Python
Q6	Yes	Even though `elif` and `if` are both keywords reserved by Python, `elif_if` is not

2. Recommended

i. Variable names should all be lowercase

ii. Words in variable names should be separated by a single underscore (_)

iii. Variable names should be concise while being sufficiently descriptive

iv. Generally, should not start with an underscore (_)

`age`, `full_name`, `height1` are all acceptable variable names in Python.

Exercise 1.2-4: Recommended Variable Names

```
Should you be using the following variable names (Yes/No)?

Q1. my_friends_best_friends_cats_name
Q2. myName
Q3. cpu_name
Q4. device__name
Q5. animal_type
```

Answer

Question	Answer	Explanation
Q1	No	Variable name is to wordy **Better:** cat_name, name
Q2	No	1. Should not contain a capital letter 2. Words should be separated using one underscore **Better:** my_name, name
Q3	Yes	Specifies what 'name' it is referring to
Q4	No	Words should be separated using one underscore **Better:** device_name
Q5	Yes	Specifies what 'type' it is referring to

2.2. Operators and Expressions

2.2.1. Expressions

An **expression** is a line of code that consists of values, variables, and/or operators. An operator is used to manipulate the values of the operands. In the expression `a + b`, `+` is the operator and `a` and `b` are the operands.

Examples of an expression

```
1    3
2    x
3    x + 3
```

2.2.2. Operators

2.2.2.1. Arithmetic Operators

Operator	Example	Description
`+` (Unary)	`+x`	Signifies that the value/variable that follows `+` is a positive number
`+` (Binary)	`x + y`	Adds the two operands together

`+` (Strings)	`str1 + str2`
	`'Hello' + 'World'`
	⇒ `'HelloWorld'`
		Concatenates the second string to then end of the first string

`-` (Unary)	`-x`	Signifies that the value/variable that follows `-` is a negative number
`-` (Binary)	`x - y`	Subtracts the right hand operand from the left hand operand
`*`	`x * y`	Multiples the two operands
`*` (Strings)	`str * n` `'a' * 4` ⇒ `aaaa`	Repeats the string operand by the number of times specified by the right hand operand
`/`	`x / y` `3 / 2` ⇒ `1.5`	Divides the left hand operand by the right hand operand
`**`	`x ** y`	Raises the left hand operand to the power of the right hand operand

`%`	`x % y`
	`4 % 2` ⇒ `0`
	`5 % 3` ⇒ `2`
		Evaluates the remainder when left hand operand is divided by the right hand operand

`//`	`x // y` `3 // 2` ⇒ `1`	Performs a floor division, which means the portion after the decimal point of the quotient is discarded. 3 divided by 2 is 1.5, so `3 // 2` discards `.5` and returns `1`.

arithmetic_operators_ex.py

```
1    x, y = 10, 20
2
3    c = x + y
4    print('x + y =', c)
5
6    c = x - y
7    print('x - y =', c)
8
9    c = x * y
10   print('x * y =', c)
11
12   c = x / y
13   print('x / y =', c)
14
15   c = x ** y
16   print('x ** y =', c)
17
18   c = x % y
19   print('x % y =', c)
20
21   c = x // y
22   print('x // y =', c)
23
24   bool = True
25   c = x + bool
26   print('x + bool =', c)
```

Output

```
x + y = 30
x - y = -10
x * y = 200
x / y = 0.5
x ** y = 100000000000000000000
x % y = 10
x // y = 0
x + bool = 11
```

Additional Explanation

Most of the examples are self-explanatory. However, lines 24-26 seem pretty confusing. How can we add a boolean value to an integer value? Remember that Python can interpret boolean values as either `False`/`True` or `0`/`1`. If an operator is used on a numerical operand and a boolean operand, Python will convert the boolean value to its corresponding numerical value.

`x` is `10` and `bool` is `True`. Since line 25 is adding a boolean value to a numerical value, Python converts `bool` to `1`. As a result, `11` is stored in `c`.

string_operators_ex.py

```
1    str1, str2 = 'Hello', 'World'
2
3    s = str1 + str2
4    print('str1 + str2 =', s)
5
6    s = str1 * 3
7    print('str1 * 3 =', s)
```

Output

```
str1 + str2 = HelloWorld
str1 * 3 = HelloHelloHello
```

Additional Explanation

Line 3 concatenates `'World'` to the end of `'Hello'`, resulting in the string `'HelloWord'`.

Line 6 repeats `'Hello'` 3 times, resulting in the string `'HelloHelloHello'`.

Exercise 2.2-1: Arithmetic Operators

```
Evaluate the following expressions.

Q1. 2 ** 10
Q2. (10 + 40) / (12 / 3)
Q3. (3 * 2) % 4
Q4. (True * 10) // 3
Q5. ('a' + 'bc') * 5
```

Answer

Q1. 1024

Explanation	2 to the power of 10 is 1024

Q2. 12.5

Explanation	(10 + 40) / (12 / 3) = 50 / (12 / 3) = 50 / 4 = 12.5

Q3. 2

Explanation	(3 * 2) % 4 = 6 % 4 = 2 When 6 is divided by 4, the remainder is 2.

Answer	
Q4. 3	
Explanation	(True * 10) // 3 = 10 // 3 = 3 `(True * 10)` is equivalent to `(1 * 10)` which evaluates to `10`. 10 divided by 3 is 3.333... Floor division discards the `.333⋯` part and yields `3`.
Q5. 'abcabcabcabcabc'	
Explanation	('a' + 'bc') * 5 = 'abc' * 5 = 'abcabcabcabcabc'

2.2.2.2. Relational Operators

These operators are used to compare the two operands and return their relationship. In programming, such relationship will always be a boolean value: `True` or `False`.

Operator	Example	Description
`==`	`x == y`	If the two operands are equal in value, then the expression evaluates to True.
`!=`	`x != y`	If the two operands are not equal in value, then the expression evaluates to True.
`>`	`x > y`	If the left hand operand is greater than the right hand operand, then the expression evaluates to True.
`<`	`x < y`	If the left hand operand is less than the right hand operand, then the expression evaluates to True.
`>=`	`x >= y`	If the left hand operand is greater than or equal to the right hand operand, then the expression evaluates to True.
`<=`	`x <= y`	If the left hand operand is less than or equal to the right hand operand, then the expression evaluates to True.

equality_operators_ex.py

```
x, y, z = 10, 20, 10
str1, str2 = 'aa', 'ab'

print('(x == z) =>', x == z)
print('(str1 == str2) =>', str1 == str2)
print('(x == str1) =>', x == str1)

print('(x != z) =>', x != z)
print('(str1 != str2) =>', str1 != str2)
print('(x != str1) =>', x != str1)
```

Output

```
(x == z) => True
(str1 == str2) => False
(x == str1) => False
(x != z) => False
(str1 != str2) => True
(x != str1) => True
```

Additional Explanation

Lines 6 and 10 show that we can compare the equality of a numerical value to a string value. Comparing the equality of a number and a string will always evaluate to `False`.
Two strings are considered equal if and only if the two strings contain the same letters in the same order.

comparison_operators_ex.py

```
 1    print('(x > y) =>', x > y)
 2    print('(x > z) =>', x > z)
 3    print('(str1 > str2) =>', str1 > str2)
 4
 5    print('(x < y) =>', x < y)
 6    print('(x < z) =>', x < z)
 7    print('(str1 < str2) =>', str1 < str2)
 8
 9    print('(x >= y) =>', x >= y)
10    print('(x >= z) =>', x >= z)
11    print('(str1 >= str2) =>', str1 >= str2)
12
13    print('(x <= y) =>', x <= y)
14    print('(x <= z) =>', x <= z)
15    print('(str1 <= str2) =>', str1 <= str2)
```

Output

```
(x > y) => False
(x > z) => False
(str1 > str2) => False
(x < y) => True
(x < z) => False
(str1 < str2) => True
(x >= y) => False
(x >= z) => True
(str1 >= str2) => False
(x <= y) => True
(x <= z) => True
(str1 <= str2) => True
```

Additional Explanation

Lines 3 and 7 checks if `str1` is either greater or less, respectively, than `str2`. The relevant magnitude of two strings are compared lexicographically (using ASCII values).

Unlike the equality operators, if we try to use the comparison operators to compare the relationship between a number and a string, Python will throw an error. For example, `10 > 'hello'` or `True <= 'world'` will cause the Python interpreter to throw an error.

comparison_boolean_ex.py

```
1    bool = True
2    print('(x > bool) =>', x > bool)
3    print('(x <= bool) =>', x <= bool)
```

Output

```
(x > bool) => True
(x <= bool) => False
```

Additional Explanation

As explained before, numerical values can be compared to boolean values. Python will convert the boolean value to its corresponding numerical value. Therefore, essentially, line 29 is the same as writing `10 > 1`, which evaluates to `True`, and line 30 is the same as writing `10 <= 1`, which evaluates to `False`.

Exercise 2.2-2: Relational Operators

```
Evaluate the following expressions.

Q1. 'A' > 'a'
Q2. 'A' > '1'
Q3. 'catc' + 'atc' + 'at' == 'cat' * 3
Q4. 33 // 3 < 9 + 2
Q5. 2 ** 10 >= 10 ** 3
```

Answer

Q1. False

Explanation	Uppercase letters have lower ASCII values than lowercase letters.

Q2. True

Explanation	Numbers (represented as strings) have lower ASCII values than alphabets (both lower and uppercase).

Q3. True

Answer	
Explanation	'catc' + 'atc' + 'at' == 'cat' * 3 ⇒ 'catcatcat' == 'catcatcat ⇒ True
Q4. False	
Explanation	33 // 3 < 9 + 2 ⇒ 11 < 11 ⇒ False
Q5. True	
Explanation	2 ** 10 >= 10 ** 3 ⇒ 1024 >= 1000 ⇒ True

2.2.2.3. Logical Operators

Whereas **Relational Operators** compared the *values* of the two operands, **Logical Operators** are mainly used to manipulate two boolean values. The resulting value will always be a boolean value. In Python, there are 3 logical operators, which are `and`, `or`, and `not`.

Let's think of the `and` and `or` operators as function that takes in two inputs, the two operands as inputs to the function, and the result as the output. The `and` operator outputs `True` if and only if the two inputs are `True`. The `or` operator outputs `True` if and only if at least one of the two inputs is `True`.

Unlike the two operators above, the `not` operator is unary, meaning it takes only one operand. The `not` operator negates the boolean value. So if the input is `True`, the output will be `False`, and vice versa.

Input		Output		
X	**Y**	**X and Y**	**X or Y**	**not X**
`False`	`False`	`False`	`False`	`True`
`False`	`True`	`False`	`True`	`True`
`True`	`False`	`False`	`True`	`False`
`True`	`True`	`True`	`True`	`False`

logical_operators_ex.py

```
1    a, b, c = 10, 20, 10
2
3    x = a > b
4    y = a == c
5    print('x is', x, 'and y is', y)
6    print('x and y =>', x and y)
7
8    x = a <= b
9    y = a > b
10   print('x is', x, 'and y is', y)
11   print('x or y =>', x or y)
12
13   x = a < b or b == c
14   print('x is', x)
15   print('not x =>', not x)
```

Output

```
x is False and y is True
x and y => False
x is True and y is False
x or y => True
x is True
not x => False
```

Additional Explanation

Lines 1 - 11 should be pretty straightforward if you look at it step-by-step.

But Line 13 may be a bit confusing at first glance. In one statement, there are both relational and logical operators. In such cases, always evaluate the relational operations first, then the logical operations. Order of operations will be covered in more depth later.

Exercise 2.2-3: Logical Operators

```
Evaluate the following expressions.

Q1. 'A' > 'a' and 'A' > '1'
Q2. not (33 // 3 < 9 + 2)
Q3. False or 2 ** 10 >= 10 ** 3
```

Answer

Q1. False

Explanation	'A' > 'a' and 'A' > '1' ⇒ False and True ⇒ False

Answer	
Q2. True	
Explanation	not (33 // 3 < 9 + 2) ⇒ not False ⇒ True
Q3. True	
Explanation	False or 2 ** 10 >= 10 ** 3 ⇒ False or True ⇒ True

2.2.2.4. Bitwise Operators

Bit operators are ones that operator on *binary numbers*. ACS Theory Lesson 6 will help in understanding this section.

Basic Explanation of Binary Numbers

It is convenient for us to count in base 10 because we have 10 fingers. However, computers do not have fingers to count with. Instead, they can 'count' in base 2 using electricity.

Suppose there is a little light bulb inside computers that lights up if there is electricity and turns off if there is no electricity. If the light bulb lights up, the computer interprets it as a `1`, and if the light bulb is off, the computer interprets it as a `0`.

The above picture is a rough demonstration of how numerical data is represented inside computers. Each `0`/`1` is called a **bit**, similar to how each number in decimal is called *digits*. Let's compare binary (base 2) numbers to decimal (base 10) numbers.

If 10110 is a decimal:

$10110_{10} = 1 * 10^4 + 0 * 10^3 + 1 * 10^2 + 1 * 10^1 + 0 * 10^0$

If 10110 is a binary number:

$10110_2 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 22_{10}$

Additional Example

Likewise, if 10110 is in base *n*:

$10110_n = 1 * n^4 + 0 * n^3 + 1 * n^2 + 1 * n^1 + 0 * n^0$

NOTE

In Python, we can represent binary numbers by preceding the binary number with `0b`.

For example, if there is a line that says `a = 0b10110`, the value 22_{10} is stored in the variable `a`.

In addition, in computer science, numbers in base 8 and 16 are also often used. They are called *octal* and *hexadecimal* numbers, respectively.

In Python, we can represent octal numbers by preceding the octal number with `0o` and hexadecimal numbers by preceding the hexadecimal number with `0x`.

Bitwise Operators

For bitwise operators, the two operands must be integer (or boolean) values. The three bitwise operators are `&` (`and`), `|` (`or`), `^` (`xor`), and `~` (`not`).

`xor` is another logical operator (not in Python) that evaluates to `True` if the two boolean operands have different values.

Input		Output
X	Y	X xor Y
False	False	False
False	True	True
True	False	True
True	True	False

If a base-10 integer is converted to a binary number, it will contain only `0`s and `1`s. Then, bitwise operators interpret each `0`/`1` bit as `False`/`True` and performs operations shown under **3. Logical Operators**. The resulting boolean value is converted back to `0`/`1`.

Operator	Example	Description
&	`x & y`	Performs bitwise AND on `x` and `y`.
|	`x | y`	Performs bitwise OR on `x` and `y`.

Operator	Example	Description
^	x ^ y	Performs bitwise XOR on x and y.
~	~x	Performs bitwise NOT on x.

NOTE x and y are integers (or booleans)

bitwise_operators_ex.py

```
1   x, y = 5, 6  # 5 -> 101, 6 -> 110
2
3   print('x & y is', x & y, 'in base 10')
4   print('x & y is', bin(x & y), 'in base 2')
5
6   print('x | y is', x | y, 'in base 10')
7   print('x | y is', bin(x | y), 'in base 2')
8
9   print('x ^ y is', x ^ y, 'in base 10')
10  print('x ^ y is', bin(x ^ y), 'in base 2')
11
12  print('~x is', ~x, 'in base 10')
13  print('~x is', bin(~x), 'in base 2')
```

Output

```
x & y is 4 in base 10
x & y is 0b100 in base 2
x | y is 7 in base 10
x | y is 0b111 in base 2
x ^ y is 3 in base 10
x ^ y is 0b11 in base 2
~x is -6 in base 10
~x is -0b110 in base 2
```

Additional Explanation

The # symbol in Line 1 is called a *comment*. Everything that follows # is ignored by the Python interpreter. We will go more in depth under Section 2.5.1.3, “Comments”.

The `bin(x)` statement in lines 4, 7, 10, and 13 is a built-in statement in Python that outputs a string form of x (decimal value) in binary. For example, `bin(22)` will return `0b10110`.

Similarly, `oct()` and `hex()` can be used for octal and hexadecimal numbers, respectively

Let's analyze how `5 & 6` works. First, `5` is converted to 101_2 and `6` is converted to 110_2.

$$\begin{array}{r} 101_2 \\ \& \ 110_2 \\ \hline 100_2 \end{array}$$

Remember that Python treats `1` as `True` and `0` as `False`. Therefore, `1 & 1` (same as `True and True`) evaluates to `True` and `0 & 1` evaluates to `False`. Then, `True`/`False` is converted back to `1`/`0`, hence, the result. Since 100_2 is 4_{10}, `5 & 6` evaluates to `4`.

TIP	If you need to check whether a number `x` is even or not, check the following condition `x & 1 == 0`. This is slightly faster than using the arithmetic modulus operator `x % 2 == 0`. This works because a binary number is even if the last bit is `0` and odd if the last bit is `1`.

Exercise 2.2-4: Bitwise Operators

```
Evaluate the following expressions.

Q1. 24 & 1
Q2. 25 & 17
Q3. 45 | 29
Q4. 13 ^ 7
```

Answer

Q1. 0

Explanation	24 & 1 = 11000_2 & 00001_2 = 0_2 = 0_{10} If `x` is even, `x & 1` is `0`.

Q2. 17

Explanation	25 & 17 = 11001_2 & 10001_2 = 10001_2 = 17_{10}

Q3. 61

Explanation	45 | 29 = 101101_2 | 011101_2 = 111101_2 = 61_{10}

Q4. 10

Answer	
Explanation	13 ^ 7 = 1101_2 | 0111_2 = 1010_2 = 10_{10}

2.2.2.5. Bitwise Shift Operators

Bitwise shift operators also operate on binary numbers. More precisely, the left hand operand is converted to a binary number and the right hand operand is left as a decimal number. Bitwise shift operators shift the left hand operand (represented in binary) either to the left or right by the number of bits specified by the right hand operand.

Example

Shift 5_{10} to the left by 3 bits
5_{10} in binary is 101_2. Then, we move 101_2 to the left by 3 bits and fill the empty bits with 0. Therefore, the result is 101000_2, or 40_{10}.

Do you see a relationship between the two numbers? 40 is 8 (= 2^3) times 5. Do you think this is a coincidence? No, it's not.

In decimal, if we shift a number `x` to the left by `n` digits, we are multiplying `x` by 10^n. For example, 5 shifted 3 times to the left is 5000 (= `5 * 10`^{3}).

Likewise, in binary, if we shift a number `x` to the left by `n` bits, we are multiplying `x` by 2^n. Therefore, when we shifted 5 to the left by 3 bits, we got 40 (= `5 * 2`^{3}).

Shift 10_{10} to the right by 1 bit
10_{10} in binary is 1010_2. We move 1010_2 to the right by 1 bit and discard the shifted bit. Since the last '0' bit is shifted, it is discarded. Therefore, we get 101_2. If we shifted 10 to the right by 2 bits, we would have gotten 10_2.

Similar to left shift, if we shift a number `x` to the right by `n` bits, we are performing a floor division by 2^n. Therefore, when we shifted 10 to the right by 2 bits, we got 2 (= `10 // 2`^{2})

Operator	Example	Description
`<<`	`x << y`	Shifts `x` (represented in binary) to the left by `y` bits.
`>>`	`x >> y`	Shifts `x` (represented in binary) to the right by `y` bits.

bitwise_shift_operators_ex.py

```
x, y = 5, 2  # 5 -> 101

print('x << y is', x << y, 'in base 10')
print('x << y is', bin(x << y), 'in base 2')

print('x >> y is', x >> y, 'in base 10')
print('x >> y is', bin(x >> y), 'in base 2')
```

Output

```
x << y is 20 in base 10
x << y is 0b10100 in base 2
x >> y is 1 in base 10
x >> y is 0b1 in base 2
```

TIP If you ever need to multiply or divide a number by a power of 2, using bitwise shift operators will be slightly faster than using arithmetic operators.

Exercise 2.2-5: Bitwise Shift Operators

```
Evaluate the following expressions.

Q1. 2 << 9
Q2. 45 >> 3
Q3. 9 << 2
Q4. 64 >> 6
```

Answer

Q1. 1024

Explanation	2 << 9 = 10_2 << 9 = $10\_000\_000\_000_2$ = $2^{10}{}_{10} = 1024_{10}$ Equivalent to $2 * 2^9$

Q2. 5

Explanation	45 >> 3 = 101101_2 >> 3 = $101_2 = 5_{10}$ Equivalent to $45 // 2^3$

Q3. 36

Answer	
Explanation	9 << 2 = 1001_2 << 2 = 100100_2 = 36_{10} Equivalent to 9 * 2^2
Q4. 1	
Explanation	64 >> 6 = 1000000_2 >> 6 = 1_2 = 1_{10} Equivalent 64 // 2^6

2.2.2.6. Assignment Operators

We met one assignment operator so far, which was `=`. The other assignment operators we are going to explore is a combination of `=` and assignment operators.

Operator	Example	Description
`=`	`x = y`	Stores the value `y` in the variable `x`.
`+=`	`x += y`	Same as `x = x + y`.
`-=`	`x -= y`	Same as `x = x - y`.
`*=`	`x *= y`	Same as `x = x * y`.
`/=`	`x /= y`	Same as `x = x / y`.
`**=`	`x **= y`	Same as `x = x ** y`.
`%=`	`x %= y`	Same as `x = x % y`.
`//=`	`x //= y`	Same as `x = x // y`.

assignment_operators_ex.py

```
x, y = 10, 20

x += y
print('After x += y, x is', x)

x = 10
x -= y
print('After x -= y, x is', x)

x = 10
x *= y
print('After x *= y, x is', x)

x = 10
x /= y
print('After x /= y, x is', x)

x = 10
x **= y
print('After x **= y, x is', x)

x = 10
x %= y
print('After x %= y, x is', x)

x = 10
x //= y
print('After x //= y, x is', x)
```

Output

```
After x += y, x is 30
After x -= y, x is -10
After x *= y, x is 200
After x /= y, x is 0.5
After x **= y, x is 100000000000000000000
After x %= y, x is 10
After x //= y, x is 0
```

Exercise 2.2-6: Assignment Operators

Evaluate the final value stored inside x.

```
x = 5
x **= 3
x //= 4
x %= 11
x -= 4
```

Answer

Q1. 5

Explanation

```
x = 5
x **= 3   # x = 125
x //= 4   # x = 31
x %= 11   # x = 9
x -= 4    # x = 5
print(x)  # will print 5
```

2.2.2.7. Ternary Operators

As the name suggests, ternary operators have three operands in the following form:

```
value1 if boolean_condition else value2
```

As the above code snippet shows, `if-else` is the ternary operator while `value1`, `boolean_condition`, and `value2` are the three operands. If the `boolean_condition` evaluates to `True`, the operator returns `value1` and returns `value2` otherwise.

ternary_operators_ex.py

```
age = 10
youth = 'young' if age <= 25 else 'old'
print('My brother is 10 years old, so he is', youth)

x, y = 5, 7
big = x if x > y else y
small = x if x < y else y
print(big, 'is greater than', small)
```

Output

```
My brother is 10 years old, so he is young
7 is greater than 5
```

Exercise 2.2-7: Ternary Operators

```
Q1. Suppose we have a variable x with an integer value. Write a program using ternary
operators to print 'Up' if x is greater than or equal to 50 and 'Down' otherwise.
Assume that variable x already has an integer value stored inside (or you can assign
an arbitrary value).

Q2. Suppose we have a variable x with an integer value. Write a program using ternary
operators to print 'Even' if x is even and 'Odd' if x is odd. Assume that variable x
already has an integer value stored inside (or you can assign an arbitrary value).

Q3. Suppose we have a variable s with a string value. Write a program using ternary
operators to print 'Before' if s is lexicographically lower than the string
'programming'. Assume that variable s already has a string value stored inside (or you
can assign an arbitrary value).
```

Sample Program

ex2_2_7.py

```
1   # Q1
2   print('Up' if x >= 50 else 'Down')
3
4   # Q2
5   print('Even' if x % 2 == 0 else 'Odd')
6   # print('Even' if x & 1 == 0 else 'Odd')
7
8   # Q3
9   print('Before' if s < 'programming' else 'After')
```

2.2.3. Type Conversion

In Section 1.1, we took a look at some of the basic data types in Python. If we add two integers together, the resulting value will be an integer type. If we add two strings together, the resulting value will be a string type.

However, if we look at the division example shown in Section 2.2.2.1, “Arithmetic Operators”, `3 / 2` resulted in `1.5`. At first glance, this does not seem important at all. But if we look closely, we can see that an operation with two *integer* operands resulted in a *floating point number*. This change of data type is called a **type conversion**.

Also, in the same section, we saw that if we add two string values, the second value is concatenated to the first value. What if we want to concatenate a number to a string? For example, if I want to

say `I am 12 years old`, will `'I am ' + 12 + ' years old'` work? Let's find out in the following sections.

2.2.3.1. Implicit Type Conversion

What does the word 'implicit' mean?

Implicit
1. implied though not plainly expressed
2. capable of being understood from something else though unexpressed

In other words, if something is expressed *implicitly*, others will be able to assume the meaning without a straightforward explanation. As such, implicit type conversion is one that Python assumes and automatically does for us.

`3 / 2` → `1.5` is an example of implicit type conversion in Python. In some other languages, like Java or C/C++, `3 / 2` will yield `1`. Because it is an operation involving two integer values, the resulting value is also an integer value. But Python will automatically convert, or implicitly convert, integer into a *float* data type.

Another example of an implicit type conversion in Python is operations involving booleans and numbers. We have seen that if boolean values are used in arithmetic operations, Python will treat them as either `0` (if `False`) or `1` (if `True`). Python is implicitly converting boolean values to integers (or floats).

In Python, the data types have a rank. Some of them are shown below.

boolean < integer < float

We can see that boolean is of lower rank than numbers, and integer is of lower rank than floating point numbers. When Python implicitly converts data types, it will always convert from lower to higher rank, in order to ***avoid data loss***.

Exercise 2.3-1: Implicit Type Conversion

```
What are the results of the following expressions?
Also check the data type of the result using type() statement if applicable (read
Section 1.1 if you forgot what type() statement is).

Q1. 10 / 4
Q2. 10 // 4
Q3. 314 / 1
Q4. True + 12
Q5. False + True
Q6. '100 + 1 is ' + 101
```

Answer

Q1. 2.5, <class 'float'>

Answer	
Explanation	Plain division in Python will always yield a *float* data type. Since `10 / 4 = 2.5`, Python yields `2.5`.
Q2. 2, <class 'int'>	
Explanation	Floor division discards the numbers after the decimal point and only returns the integer value. Since `10 / 4 = 2.5`, floor division will discard `.5` and return `2`.
Q3. 314.0, <class 'float'>	
Explanation	As explained before, division in Python will always result in a *float* data type, even if it is a division without a remainder.
Q4. 13, <class 'int'>	
Explanation	Since `True` is equal to `1` in Python, `True + 12` yields the integer value `13`.
Q5. 1, <class 'int'>	
Explanation	`False + True` is equivalent to saying `0 + 1`, which evaluates to the integer value `1`.
Q6. TypeError: unsupported operand type(s) for +: 'int' and 'str'	
Explanation	Python will not implicitly convert numbers to strings, unlike some other languages. Since Q6 attempted to concatenated the integer `101` to a string, Python throws an error (name of error is not important for now)

2.2.3.2. Explicit Type Conversion

If you attempted Exercise 2.3-1, you may have seen that Q6. `'100 + 1 is ' + 101` did not work. This means that concatenating a number to a string by using the plus (+) operator does not work. Then what should we do in these kinds of situations?

We can use the built-in `str()`, `int()`, and `float()` functions.

`str(x)`

`str(x)` function will convert `x` into its corresponding string value.

```
1    str(10)                          # yields '10'
2    type(str(10))                    # yields '<class 'str'>'
3    str(3.14)                        # yields '3.14'
4    str(1 + 2 + 3)                   # yields '6'
5    'I am ' + str(15) + ' years old'  # yields 'I am 15 years old'
```

`int(x)`

`int(x)` will convert `x` into its corresponding integer value.

```
int('13')        # yields 13
type(int('13'))  # yields '<class 'int'>'
int('  14  ')    # yields 14
int(3.14)        # yields 3
10 + int('15')   # yields 25
```

`int(x, b)`

`int(x, b)` will interpret the string value `x` as a number in base `b` and yield the corresponding integer value.

```
int('10', 2)    # yields 2
int('1011', 2)  # yields 11
int('13', 8)    # yields 11
int('b', 16)    # yields 11
```

`float(x)`

`float(x)` will convert `x` into its corresponding float value.

```
float(10)          # yields 10.0
type(float(10))    # yields '<class 'float'>'
float('3.14')      # yields 3.14
float(' 1  ')      # yields 1.0
1.0 + float('2.0')  # yields 3.0
```

Explicit type conversion can also be called type casting because we are *casting*, or forcefully imposing, another data type to the values.

Exercise 2.3-2: Explicit Type Conversion

```
What are the results of the following expressions?
Also check the data type of the result using type() statement if applicable (read
Section 1.1 if you forgot what type() statement is).

Q1. int('13' + '14')
Q2. int('13' + ' 14')
Q3. int(13 + '14')
Q4. str(123) + '456'
Q5. 123 + int('456')
Q6. str(int('1010', 2))
Q7. 4 / 2 == float(2)
```

Answer

Q1. 1314, <class 'int'>

Answer	
Explanation	First, the expression inside the parentheses is evaluated. `'13' + '14'` evaluates to `'1314'`, which is then converted to its corresponding integer value, which is `1314`.
Q2. 2, ValueError: invalid literal for int() with base 10: '13 14'	
Explanation	The expression inside the parentheses evaluates to `'13 14'` which is not a valid integer value. So Python throws an error.
Q3. TypeError: unsupported operand type(s) for +: 'int' and 'str'	
Explanation	As explained before, a number cannot be added to a string value, so Python throws an error.
Q4. '123456', <class 'str'>	
Explanation	The integer value `123` is converted to a string. Then `'123'` is added to `'456'`, resulting in the string value `'123456'`.
Q5. 579, <class 'int'>	
Explanation	The string value `'456'` is converted to an integer. Then the two numbers are added together, yielding the integer value 579.
Q6. '10', <class 'str'>	
Explanation	`'1010'` is interpreted as a binary number, which is `10` in decimal. So `str(int('1010', 2))` evaluates to `str(10)`, which yields `'10'`.
Q7. True, <class 'bool'>	
Explanation	The left hand side of the equation evaluates to `2.0`, and the right hand side also evaluates to `2.0`. So the equation is `True`.

2.2.4. Order of Operations

Now that we have looked at seven types of operators, if you saw the following statement, how would you solve it?

```
not (1 + 3 * 2) >= 3 ** 3 * 2 // 6 and 35 & 12 < 25
```

It is not immediately clear which operation you should start solving. This is why Python designated the order in which the operations will be solved in, called **Order of Operations** or **Operator Precedence**.

Priority[1]	Operator	Example	Name
1	(*expressions...*)	`(a + b - c)`	Parenthesized Expression
2	`**`	`a ** b`	Exponentiation[2]

Priority[1]	Operator	Example	Name
3	`+` (unary)	`+a`	Positive (number)
	`-` (unary)	`-a`	Negation (of a number)
	`~` (unary)	`~a`	Bitwise NOT
4	`*`	`a * b`	Multiplication
	`/`	`a / b`	Division
	`//`	`a // b`	Floor division
	`%`	`a % b`	Modulus (Remainder)
5	`+`	`a + b`	Addition
	`-`	`a - b`	Subtraction
6	`<<`	`a << b`	Bitwise left shift
	`>>`	`a >> b`	Bitwise right shift
7	`&`	`a & b`	Bitwise AND
8	`^`	`a ^ b`	Bitwise XOR
9	`|`	`a | b`	Bitwise OR
10	`<`	`a < b`	Less than
	`<=`	`a <= b`	Less than or equal to
	`>`	`a > b`	Greater than
	`>=`	`a >= b`	Greater than or equal to
	`==`	`a == b`	Equal to
	`!=`	`a != b`	Not equal to
	`in`, `not in`, `is`, `is not`	*Omitted*	Membership and Identity tests (not covered)
11	`not`	`not X`	Boolean NOT
12	`and`	`X and Y`	Boolean AND
13	`or`	`X or Y`	Boolean OR
14	`=`	`a = b`	Assignment operator
	`+=`, `-=`, `*=` ...	*Omitted*	Shorted assignment operators

1 1 has highest priority while 13 is of the lowest priority. If there are multiple operators of the same priority, they are evaluated from left to right.

2 `**` (Exponentiation) has lower precedence than the arithmetic/bitwise unary operators to its right. For example, `2**-1` is `0.5` (Do `-1` first. Then calculate `2 ** (-1)`)

NOTE	Like mathematics, arithmetic expressions in programming also follow the **PEMDAS** (Parentheses-Exponent-Multiplication-Division-Addition-Subtraction) rule.

Exercise 2.4-1: Order of Operations

```
Q1. 10 - 6 + 3 * 4 // 4
Q2. -5 - --6 * 9 / (2 - 1) * 3
Q3. not (1 + 3 * 2) >= 3 ** 3 * 2 // 6 and 35 & 12 < 25
```

Answer

Q1. 7

Explanation	10 - 6 + 3 * 4 // 4 = 10 - 6 + `3 * 4` // 4 = 10 - 6 + `12 // 4` = `10 - 6` + 3 = `4 + 3` = `7`

Q2. -167.0

Explanation	-5 - --6 * 9 / (2 - 1) * 3 = -5 - --6 * 9 / `(2 - 1)` * 3 = `-5` - --6 * 9 / 1 * 3 = -5 - `--6` * 9 / 1 * 3 = -5 - `--6` * 9 / 1 * 3 = -5 - `6 * 9` / 1 * 3 = -5 - `54 / 1` * 3 = -5 - `54.0 * 3` = `-5 - 162.0` = `-167.0`

Q3. `True`

Explanation	not (1 + 3 * 2) >= 3 ** 3 * 2 // 6 and 35 & 12 < 25 = not `(1 + 3 * 2)` >= 3 ** 3 * 2 // 6 and 35 & 12 < 25 = not 7 >= `3 ** 3` * 2 // 6 and 35 & 12 < 25 = not 7 >= `27 * 2` // 6 and 35 & 12 < 25 = not 7 >= `54 // 6` and 35 & 12 < 25 = not 7 >= 9 and `35 & 12` < 25 = not `7 >= 9` and 0 < 25 = not False and `0 < 25` = `not False` and True = `True and True` = `True`

2.3. Printing

When we are programming, we may want to display information on screen, whether it is to check the output or debug. Since computers cannot talk, displaying information on screen is sometimes a convenient way to communicate with the computer while coding.

To achieve this, we can use the built-in `print()` function. We have seen this function a few times in previous examples and exercises. Now, let's take a deeper look at this function.

2.3.1. print(*x*)

`print(x)` will first convert `x` into its corresponding string value. If `x` is an expression, the expression will be evaluated first, then converted to a string value. Then, the string value will be printed on screen, and the cursor moves to the next line.

print_ex.py

```
1    print('Hello World')
2    print(1 + 2 + 3)
3    print('I have ' + str(3) + ' cats')
```

Output

```
Hello World
6
I have 3 cats
```

Additional Explanation

`print('Hello World')` displays `Hello World` on screen, then the cursor is moved to the next line. The second statement, `print(1 + 2 +3)` prints `6` from where the cursor is placed, which is the line after `Hello World`. That is why the output is separated by lines.

If the `print()` statement did not move the cursor to the next line, the output will look like:

```
Hello World6I have 3 cats
```

Exercise 3.1-1: Print Statements

```
Write a Python program to print the following sentences using 3 print() statements.

Python is fun.
To become better at Python,
you should spend at least 1 hour each day practicing.
```

Sample Program

ex3_1_1.py

```
1    print('Python is fun.')
2    print('To become better at Python,')
3    print('you should spend at least 1 hour each day practicing.')
```

2.3.1.1. Escape Characters

What if we want to print single quotations with the `print()` statement? Can we write the following?

```
print('I said, 'Hello'')
```

To achieve this task, we can use a special character, called backslash (`\`). Below lists some of the most commonly used escape characters (characters used with a backslash). If you want to learn more, Google is your best friend.

Escape Character	Name	Description
`\n`	newline	Moves the cursor to the next line
`\t`	tab	Prints the ASCII Horizontal Tab (TAB)
`\'`	Single quote	Prints a single quote
`\"`	Double quote	Prints a double quote

escape_characters_ex.py

```
1    print('Hello World\n')
2    print('\ta')
3    print('I said, \'Hello\'')
4    print('I said, \"Hello\"')
```

Output

```
Hello World

        a
I said, 'Hello'
I said, "Hello"
```

Additional Explanation

The output from Line 1 results in a blank line. Why is this so? If we include the escape character `'\n'`, the escape character will move the cursor to the next line once. Then, the `print()` statement will move the cursor to the next line once more time, causing a blank line.

NOTE

Since we use single quotes to represent strings, using double quotes without the backslash character is also fine. For example, `print('I said, "Hello"')` will also print `I said, "Hello"`.

The reason why there is a `\"` is because, in Python, we can represent strings with double quotes as well (although this is discouraged). If we used double quotation marks, then we would have to write `print("I said, \"Hello\"")`, in order to print `I said, "Hello"`.

Similarly, we can use single quotes without the backslash character if we represent strings with double quotation marks. For example, `print("I said, 'Hello'")` will print `I said, 'Hello'`.

Exercise 3.1-2: Escape Characters

```
Write a Python program to print the following sentences using 3 print() statements and
escape characters.

        The start of a paragraph should be indented.
Can I write one sentence in
two lines using one print statement?
We can print quotation marks, like ' and ", using print statements.
```

Sample Program

ex3_1_2.py

```
1    print('\tThe start of a paragraph should be indented.')
2    print('Can I write one sentence in\ntwo lines using one print statement?')
3    print('We can print quotation marks, like \' and \", using print statements.')
```

2.3.2. print(*x1*, *x2*, ...)

This is the statement that was used often in previous exercises. The values to be printed are separated by commas. Then Python will print all of the values separated by a single whitespace.

comma_print_ex.py

```
1    print(1, 2, 3)
2    print('I have', 3, 'cats')
```

Output

```
1 2 3
I have 3 cats
```

Additional Explanation

Line 1 is trying to print three integer values. 1, 2, and 3 are converted into strings and printed, separated by a single whitespace. It is equivalent to saying `print(str(1) + " " + str(2) + " " + str(3))`.

Line 2 shows that you can print strings together with numbers by listing them with commas. This is a clearer and more concise alternative to writing `print('I have ' + str(3) + ' cats')`.

Exercise 3.2-1: Printing Multiple Values

```
Write a Python program to print the following sentences using 2 print() statements.

I want to have 2 puppies
1 plus 1 is 2
```

Sample Program

ex3_2_1.py

```
1    print('I want to have', 2, 'puppies')
2    print(1, 'plus', 1, 'is', 2)
```

2.3.3. print(*x*, end=*s*)

`print(x, end=s)` will concatenate `s` to the end of `x` and print the resulting value without moving the cursor to the next line.

It was mentioned before that the `print()` statement automatically moves the cursor to the next line. This was a very simplified way of explaining the `print()` statement. In reality, when we use the `print()` statement, `s` is set to `'\n'` by default. Therefore, `print(x)` is equivalent to `print(x, end='\n')`. In any case where the value for `end` is not specified, it is set to `'\n'` by default.

end_print_ex.py

```
1    print('Hello World', end='\n')
2    print('1 2 3', end='\n\n')
3    print('Hello', end='World ')
4    print('The line does not change.', end='Weird!\n')
```

Output

```
Hello World
1 2 3

HelloWorld The line does not change.Weird!
```

Additional Explanation

Line 1 is equivalent to saying `print('Hello World')`.

Line 2 concatenates two newline characters to the end of `'1 2 3'`, so a blank line is created.

Line 3 concatenates `'World '` to the end of `'Hello'` (which results in `'HelloWorld '`) and is printed without moving the cursor to the next line. Therefore, the `print()` statement in Line 4 starts from the end of `'HelloWorld '`.

Exercise 3.3-1: End Strings

```
Write a Python program to print the following sentences using 3 print() statements.

This print statement has no line change.This is the next print statement ending in **
The last print statement ending in HelloWorld
```

Sample Program

ex3_3_1.py

```
1    print('This print statement has no line change.', end='')
2    print('This is the next print statement ending in ', end='**\n')
3    print('The last print statement ending in ', end='HelloWorld\n')
```

2.3.4. print(*x*..., sep=*s*)

`print(x..., sep=s)` will separate the given values `x...` by the given string `s`. For example, if we say `print(x1, x2, x3, sep=s)`, it equivalent to saying `print(x1 + s + x2 + s + x3)`, assuming `x1`, `x2`, and `x3` are all string values.

Remember that if we list multiple values with commas, the values are printed separated by a single whitespace. This is because `sep` value is set to `' '` by default.

sep_print_ex.py

```
1    print('Hello', 'World', sep=' ')
2    print(1, 2, 3, sep='+')
3    print('apples', 'banana', 'oranges', sep=', ')
4    print('Nothing happens if there is only one value', sep='What would happen?')
```

Output

```
Hello World
1+2+3
apples, banana, oranges
Nothing happens if there is only one value
```

Additional Explanation

Line 1 is equivalent to saying `print('Hello', 'World')`.

Line 2 is equivalent to saying `print(str(1) + '+' + str(2) + '+' + str(3))`.

Nothing happens in Line 4 because there is only one value. This `print()` statement joins multiple values together separated by the `sep` value. If there is only one value, there is nothing to join together.

Exercise 3.4-1: Separating Characters

```
Write a Python program to print the following sentences using 2 print() statements.

We can separate values by commas like A, B, C.
Parentheses->Exponent->Multiplication->Division->Addition->Subtraction.
```

Sample Program

ex3_4_1.py

```
1    print('We can separate values by commas like A', 'B', 'C.', sep=',')
2    print('Parentheses', 'Exponent', 'Multiplication', 'Division', 'Addition',
'Subtraction.', sep='->')
```

2.4. Input

So far, to store a value inside a variable, we specified it directly in the program. For example, if we wanted to store the value `10` in the variable `a`, we would have to write a line in our program as such: `a = 10`. However, say we have a program to store personal data. We would have hundreds or thousands of users using our program. To store each user's information, such as their name, age, and gender, we cannot be updating our program directly each time. We need the users to input their own information and a way for our program to interpret and store the users' inputted information.

To do this, we use the built-in `input()` function.

2.4.1. input(*prompt*)

To see what `input(prompt)` function does, let's take a look at an example.

input_ex1.py

```
1    name = input('What is your name: ')
2    print('User\'s name is ' + name)
```

CLI

```
What is your name: John Doe
User's name is John Doe
```

NOTE CLI (command line interface) is a platform where you can interact with your program.

Explanation

`input('What is your name: ')` in Line 1 prints the prompt message `'What is your name: '` on screen (without a newline character).

The user types his/her input on the CLI. In this example, the user typed `John Doe` as the input and pressed enter (which is recognized as a newline character).

When Python sees the newline character, Python takes all the characters from after the prompt message and before (and not including) the newline character and stores it in the variable `name` as a string type. Therefore, after the user presses enter, `'John Doe'` is stored in the variable `name`.

Line 2 prints the user's name along with some explanatory text.

input()` treats all user inputs as a string. In order to treat them as other data types, such as integers or floats, we have to type cast them (covered under Section 2.2.3.2, "Explicit Type Conversion").

input_ex2.py

```
1    # Input section
2    name = input('Write your name: ')
3    age = int(input('Write your age: '))
4    letter = input('Write a letter: ')
5    pi_approx = float(input('Write the pi approximate: '))
6
7    # Output section
8    print('Your name is', name)
9    print('You are', age, 'years old')
10   print('Your letter is', letter)
11   print('Pi is approximately', pi_approx)
```

CLI

```
Write your name: John Doe
Write your age: 24
Write a letter: A
Write the pi approximate: 3.14159
Your name is John Doe
You are 24 years old
Your letter is A
Pi is approximately 3.14159
```

Additional Explanation

Lines 3 and 5 are casting the user input into an integer and float, respectively. In the example program above, there is no reason to type cast the user input, but in other situations, you might need to.

Exercise 4.1-1: Input

```
Q1. Suppose there is a new amusement park built in your city. You are asked to make a
program that calculates the ticket price. The price of each ticket is $5. Prompt the
user how many tickets they need and calculate the price for them.

Example interaction:
How many tickets do you need: 5
Total ticket price is $25

Q2. The ticket pricing has changed. Now, the ticket price for children under the age
of 7 is $3, and for everyone else, it is $5. Write a program that calculates the
ticket price.
(Hint: use more than one input() statement with meaningful prompt messages)
```

Sample Program

ex4_1_1.py

```
1    # Q1
2    num_ticket = int(input('How many tickets do you need: '))
3    print('Total ticket price is $' + str(num_ticket * 5))
4
5    # Q2
6    num_child = int(input('How many children (under the age of 7) are there: '))
7    num_adult = int(input('How many adults are there: '))
8    print('Total ticket price is $' + str(num_child * 3 + num_adult * 5))
```

2.5. Tips

This section covers some tips on how you can become a better programmer.

2.5.1. Coding Style

Just like how spoken languages have grammar, programming languages have syntax. And no matter how your code is written, as long as it correctly follows the syntax, the Python interpreter will be able to understand it. However, it might not be the case for humans. If you start working, you will not be the only one looking at your code. Others would have to take a look and make corrections or give feedback on your work. Suppose you saw the following block of code:

```
A= 3
B=5

a= int(input())
b =  int(input() )

c =  a+b
d= a  *A  + b*B

print(c , d)
```

Can you tell what this snippet of code is trying to achieve? It can be an example showcasing the use of different arithmetic operators, or it can be a program to calculate something more meaningful, like ticket prices. Also, the code does not look clean overall. The following snippet of code would be more appreciated by your peers or coworkers:

```
# Calculate and display the total number of visitors coming to this zoo
# and the total ticket price.

CHILD_TICKET_PRICE = 3  # in dollars
ADULT_TICKET_PRICE = 5  # in dollars

num_child = int(input('Number of children: '))
num_adult = int(input('Number of adults: '))

total_visitor = num_child + num_adult
total_price = num_child * CHILD_TICKET_PRICE + num_adult * ADULT_TICKET_PRICE

print('Total number of visitors:', total_visitor)
print('Total ticket price:', total_price)
```

How well a code can be understood by other people is called *readability*, and Python has a set of styling guidelines to make your code more *readable* to others. Let's take a look at some of them.

2.5.1.1. Variable Naming Conventions

*** Included from Section 2.1.2.3, "Naming Conventions of Variables"**

`Capitalized`, `PascalCase`, `camelCase` are all valid variable names. However, to standardize variable names, Python has a set of rules when naming variables.

1. Not allowed

i. Cannot contain any characters other than alphabets, numbers, and underscores

ii. Cannot start with a number

iii. Cannot be a word reserved by Python (called a keyword)

Below shows all 35 keywords reserved by Python:

`and`	`del`	`from`	`None`	`True`
`as`	`elif`	`global`	`nonlocal`	`try`
`assert`	`else`	`if`	`not`	`while`
`break`	`except`	`import`	`or`	`with`
`class`	`False`	`in`	`pass`	`yield`
`continue`	`finally`	`is`	`raise`	`async`
`def`	`for`	`lambda`	`return`	`await`

2. Recommended

i. Variable names should all be lowercase

ii. Words in variable names should be separated by a single underscore (_)

iii. Variable names should be concise while being sufficiently descriptive

iv. Generally, should not start with an underscore (_)

`age`, `full_name`, `height1` are all acceptable variable names in Python.

Also, there is a difference when naming constants and variables. As covered before, constants are values that do not change, like pi and speed of light, and variables are values that can change. As shown above, variable names should be in lowercase.

Constants should follow the same rules as other variables, but they should all be capitalized.

```
4    CHILD_TICKET_PRICE = 3  # in dollars
5    ADULT_TICKET_PRICE = 5  # in dollars
```

We know that ticket prices for children and adults will not change, so we set them as constants when we write our code. When we do so, we capitalize their names to let others know that the corresponding variable is storing a constant value.

2.5.1.2. Spacing

Remember the first code example? It looked like the following:

```
1    A = 3
2    B = 5
3
4    a = int(input())
5    b =  int(input() )
6
7    c = a+b
8    d = a  *A  + b*B
9
10   print(c , d)
```

See lines 5, 7, 8, and 10? What do they all have in common? They have spacing issues.

Generally, for expressions, we separate operands and operators (except for unary operators) with one whitespace.

```
# Binary operators
operand_a operator operand_b

# Ternary operators
operand_a operator operand_b operator operand_c
```

Below are some examples.

```
# Unary operators (no space)
+x
-x
~x

# Binary operators
# Bad
x+y
x- y
x  *  y
a=6

# Good
x + y
x - y
x * y
a = 6

# Ternary operator
# Bad
x  if boolean_condition  else  y

# Good
x if boolean_condition else y
```

When we are listing things with a comma, we place the comma right after the previous element with no space and separate the next element with one space.

```
A, B, C, D, ...
```

Below are some examples.

```
# Bad
print(x,y,z)
print(x , y , z)

# Good
print(x, y, z)
```

2.5.1.3. Comments

It was briefly mentioned before that whatever comes after # is called a *comment*. A comment is not recognized by the Python interpreter and will have no effect whatsoever on your code.

Then why should we write comments? After finishing a long piece of code, if you leave it and come back a few months later, chances are, you will not be able to understand what you wrote. To help you and others understand what your code is talking about, it is necessary to write meaningful

comments.

The two types of comments that we are going to cover are *block comments* and *inline comments*.

Block Comments

What is a block? Let's look at the previous example again and find out.

```
1    # Calculate and display the total number of visitors coming to this zoo
2    # and the total ticket price.
3
4    CHILD_TICKET_PRICE = 3  # in dollars
5    ADULT_TICKET_PRICE = 5  # in dollars
6
7    num_child = int(input('Number of children: '))
8    num_adult = int(input('Number of adults: '))
9
10   total_visitor = num_child + num_adult
11   total_price = num_child * CHILD_TICKET_PRICE + num_adult * ADULT_TICKET_PRICE
12
13   print('Total number of visitors:', total_visitor)
14   print('Total ticket price:', total_price)
```

You can see that all the code in Lines 1-14 are performing one task: to calculate the total number of visitors and ticket price. Likewise, a *block* in Python can be loosely defined as a set of code that serves the same purpose or task.

Another way to think of a *block* is to check if a chunk of code has the same indentation. Later, you will learn *if-statements* and *while-loops* that will have different indentations. Those would also be containing distinct *blocks* of code.

Then what is a *block comment*? It is a comment explaining what the subsequent *code block* is trying to do.

```
1    # Calculate and display the total number of visitors coming to this zoo
2    # and the total ticket price.
```

This block comment explains what Lines 4-14 are trying to achieve. As such, meaningful block comments can help others understand what your program is doing without dissecting each line of code.

Block comments should be placed in the same indentation level as the following block. Also, each block comment should generally start with a `#` followed by a single whitespace. For example:

```
# Bad
#   Calculate and display the total number of visitors coming to this zoo
#   and the total ticket price.

    # Calculate and display the total number of visitors coming to this zoo
    # and the total ticket price.

# Good
# Calculate and display the total number of visitors coming to this zoo
# and the total ticket price.
```

Inline Comment

Like the name suggests, an *inline comment* is one that is written in the same line as the statement you are trying to explain.

```
4    CHILD_TICKET_PRICE = 3  # in dollars
5    ADULT_TICKET_PRICE = 5  # in dollars
```

This is an example of inline comments. `# in dollars` explains that the written price is measured in dollars.

As shown in the example, inline comments should be written at least two spaces from the statement you are trying to explain. Also, like block comments, they should start with a `#` followed by a single whitespace.

However, try to avoid overusing inline comments, especially if the line of code is very obvious.

```
# Bad
x = x + 1  # increment x by 1
name = input()  # Get user's name
print(name)  # print the user's name
```

2.6. Practice Problems

2.6.1. Problems

A. Variable Naming Convention

A-1 Are the following variable names acceptable? If not, explain why.

Variable	Yes/No	Explain (if applicable)
first name		
cpu_price		
_distance		

Variable	Yes/No	Explain (if applicable)
child-age		
PI		
name_of_my_new_cpu		

Answer

Variable	Yes/No	Explain (if applicable)
first name	No	Variable names cannot contain spaces. Multiple words should be separated with an underscore. **Better:** first_name, name
cpu_price	Yes	
_distance	No	It is a valid name, but variable names generally should not start with an underscore. **Better:** distance, dist
child-age	No	Variable names cannot contain hyphens. Multiple words should be separated with an underscore. **Better:** child_age, age
PI	Yes	Constant names should be all capitalized.
name_of_my_new_cpu	No	Variable name is too lengthy **Better:** cpu_name, name

B. Evaluate the displayed output

B-1

B_1.py

```
f = 100.32;
a = int(f)
s = str(f + a)

print('f =', f, 'a =', a, 's =', s)
```

Answer

Output

```
f = 100.32 a = 100 s = 200.32
```

B-2

B_2.py

```
a = 10
mystery = str(a) * 2
print(int(mystery, 2))
```

Answer

Output

```
10
```

B-3 What is the following the program doing? What would the output be if the input was 12? 28? 32? 45?

B_3.py

```
mystery_num = int(input('Mystery number: '))
mystery_condition = mystery_num % 3 == 0

print('Yes :)' if mystery_condition else 'No :(')
```

Answer

The program is checking if the inputted number is divisible by 3 (prints `Yes :)`) or not (prints `No :(`).

Output

```
Mystery number: 12
Yes :)
Mystery number: 28
No :(
Mystery number: 32
No :(
Mystery number: 45
Yes :)
```

B-4

B_4.py

```
x = 1

print(True and (3 > 4))
print(not x > 0)
print(x != 1 or x < 1)
print(x >= 0 or not x == 1)
```

Answer

Output

```
False
False
False
True
```

B-5

B_5.py

```
print(4 << 2)
print(10 & 5)
print(23 ^ 13)
print(15 | 4)
```

Answer

Output

```
16
0
26
15
```

B-6

B_6.py

```
a, b, c, d, e = 3, 2, 1, 4, 5

num1 = a * b + d / c % e
num2 = -a * (b+b) - c * d // (4 - 2) * e
num3 = (a - b) * 10 % 3 + c * d - (4 - 2)

print('num1 =', num1)
print('num2 =', num2)
print('num3 =', num3)
```

Answer

Output

```
num1 = 10.0
num2 = -22
num3 = 3
```

C. Find any errors in following Python programs

C-1

C_1.py

```
var.name = 1
print(var.name + 9)
```

Answer

```
var.name = 1          # -> var_name = 1
print(var.name + 9)   # -> print(var_name + 9)
```

C-2

C_2.py

```
int x = 1
int y = 2

print(x + y)
```

Answer

```
int x = 1  # -> x = 1
int y = 2  # -> x = 2

print(x + y)
```

C-3

C_3.py

```
num1 = input('Type first number: ')
num2 = input('Type second number: ')

print('The average of the two numbers are', (num1 + num2) / 2)
```

Answer

```
num1 = input('Type first number: ')   # -> num1 = int(input(...))
num2 = input('Type second number: ')  # -> num2 = int(input(...))

print('The average of the two numbers are', (num1 + num2) / 2)
# or print('...', (int(num1) + int(num2)) / 2)
```

D. Write your own program

D-1

You are making a program to register users in a game company. Get the user's profile and print their information in the following manner:

Example Output

```
Name: John Doe
Age: 15
E-mail: johndoe@zmail.com
Phone: 4082839123
John Doe, 15 years old, johndoe@zmail.com, 4082839123
```

Sample Program

d_1_sample.py

```
name = input('Name: ')
age = input('Age: ')
email = input('E-mail: ')
phone = input('Phone: ')
print(name, age + ' years old', email, phone, sep=', ')
```

D-2

Before the actual construction, a building company needs to find out the volume of the intended building. The building designer Sam who has to build thousands of buildings does not want to calculate volume of each building by hand. So, he plans to make a program that calculates the volume if a user inputs the width, length, height. Write this program.

Example Output

```
Width: 200
Length: 200
Height: 400
Volume: 16000000
```

Sample Program

d_2_sample.py

```
width = int(input('Width: '))
length = int(input('Length: '))
height = int(input('Height: '))

vol = width * length * height
print('Volume:', vol)
```

D-3

CSM Pizza wants to provide a differentiated service to customers. So they have decided to make pizzas in sizes that the customers want, instead of the standard small, regular, and large size. Make a program that gets the preferred radius from customers and tell them the area and the price of the pizza ($1.5 per 1 cm). Use 3.14 for the value of pi.

Example Output

```
Welcome to CSM Pizza!
Preferred radius of your pizza (in cm): 10
The area of your pizza is 314.0 cm^2
It is 15.0 dollars.
```

Sample Program

d_3_sample.py

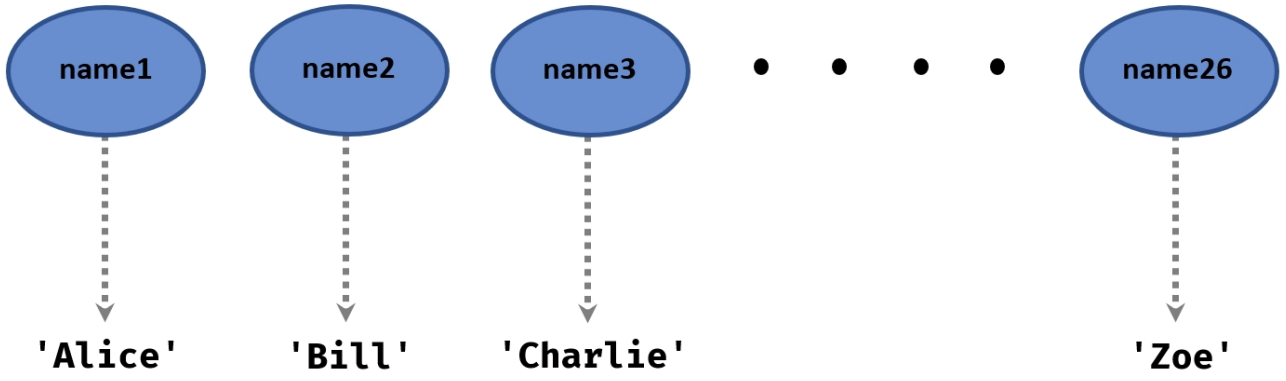
```
PI = 3.14
PRICE_PER_CM = 1.5

print('Welcome to CSM Pizza!')
radius = int(input('Preferred radius of your pizza (in cm): '))

area = PI * radius * radius  # same as PI * radius ** 2
price = PRICE_PER_CM * radius

print('The area of your pizza is', area, 'cm^2')
print('It is', price, 'dollars')
```

3. Chapter 3

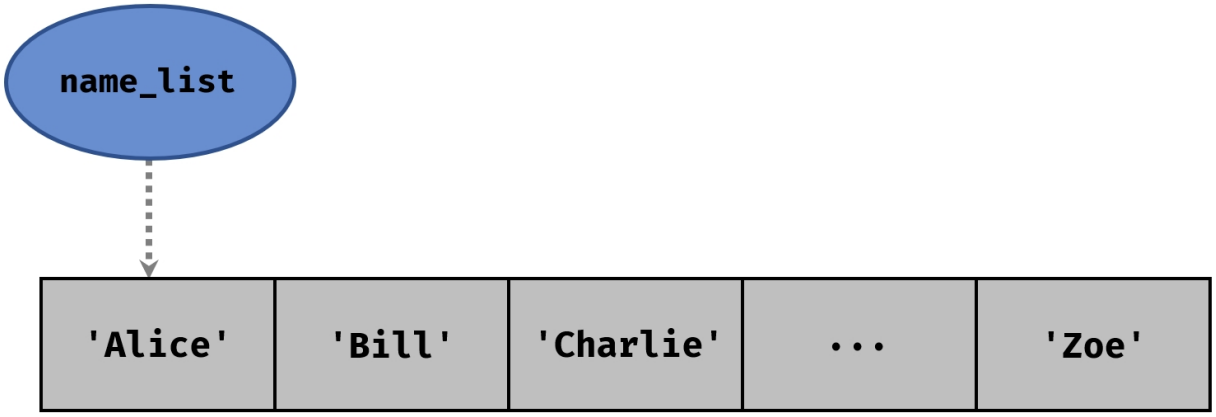
3.1. Lists

3.1.1. What is a List?

Suppose there is a school with a class size of 26 students. As a member of the IT department, it is your responsibility to organize and manage student data. So if you want to store all the students' names of each class, how would you do it? Should you make 26 `name` variables and store each student's name, like the following?



The answer is we can use a *list*. A list is a collection, or a sequence, of elements. For the above example, instead of creating 26 separate variables for each student, we can create one *list* object that can hold 26 different values, whether it be names, ages, or genders.



3.1.2. Initialization

A list is a data type supported by Python, like numbers, strings, and booleans. To create lists in Python, we use square brackets (`[]`).

List Examples

```
1    [1, 2, 3, 4]  # list of 4 integers
2    ['Alice', 'Bill', 'Charlie', 'Zoe']  # list of 4 strings
3    [1, 'Alice', 2, 'Bill']  # a list can contain values/variables of different types
```

All three lines are examples of a list in Python. As it shows, each element is separated with a comma (`,`), and the elements of a list can be of different types. We can also assign a list to a variable.

List Assignment

```
1    numbers = [1, 2, 3, 4]
2    names = ['Alice', 'Bill', 'Charlie', 'Zoe']
3    empty_list = []
```

Assigning is the same as you would do for other data types, such as numbers, strings, and booleans. If you look carefully at the variable names, you can see that they are plural (except for line 3). Generally, for names of list variables, you should use the plural form of the kind of data you are storing in that list.

List Naming Convention

```
1    # only storing one 'name'
2    name = 'Alice'
3    # storing multiple 'names'
4    names = ['Alice', 'Bill', 'Charlie', 'Zoe']
```

Python also allows you to print lists using the `print()` statement.

Printing Lists

```
1    numbers = [1, 2, 3, 4]
2    names = ['Alice', 'Bill', 'Charlie', 'Zoe']
3    print(numbers)
4    print(names)
```

Output

```
[1, 2, 3, 4]
['Alice', 'Bill', 'Charlie', 'Zoe']
```

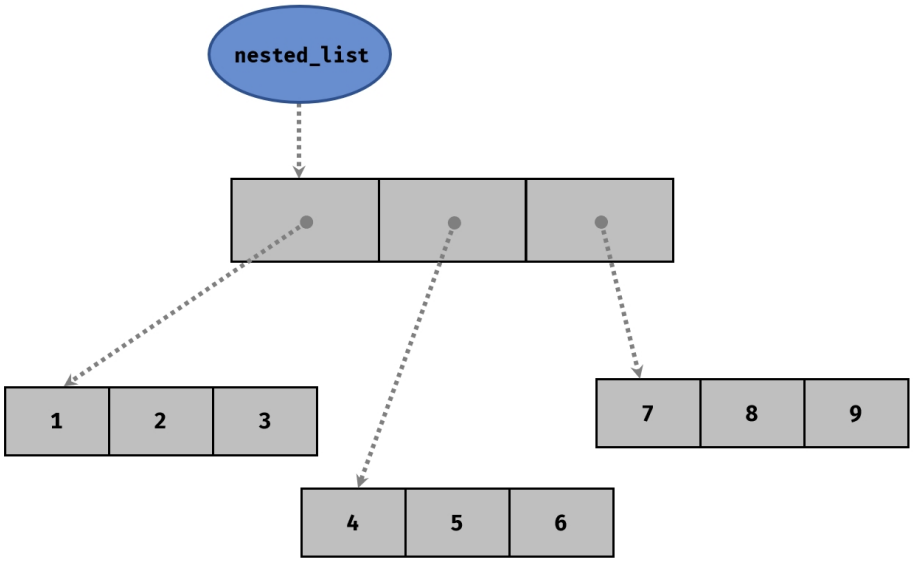
Nested Lists

Since a list can contain elements of any type, can a list contain a list? The answer is yes, and we call those *nested lists*.

Nested List Example

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

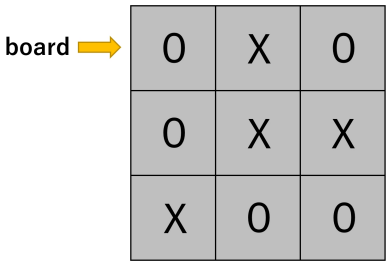
The important thing to note is that `nested_list` is a list with 3 elements (not 9), where each element is another list containing 3 elements.



We also call lists with similar structures as `nested_list` *2D lists* because they can be used to describe a 2 dimensional board. For example, we can use a 2D list to describe a tic-tac-toe board.

2D List Example

```
1    board = [['O', 'X', 'O'],
2             ['O', 'X', 'X'],
3             ['X', 'O', 'O']]
```

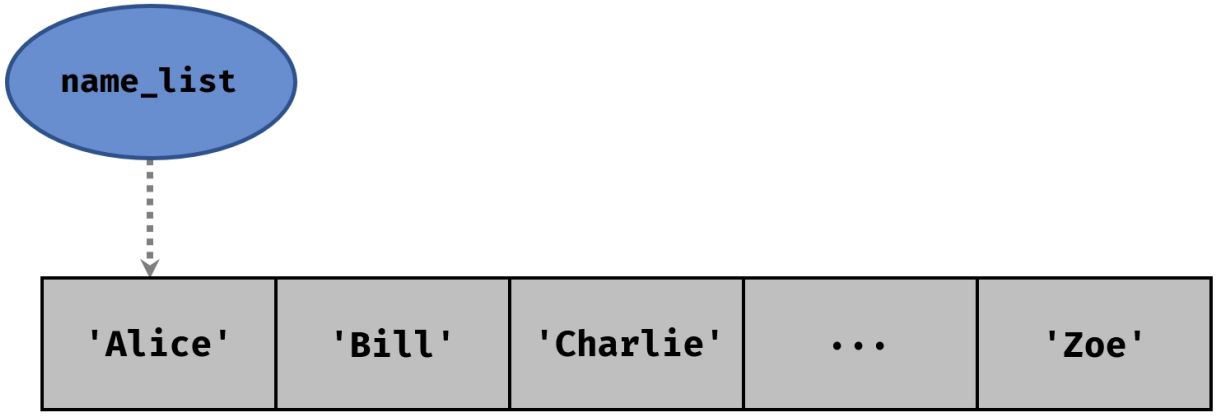


Similarly, we can have 3D, 4D, and other higher dimensional lists, but as of now, you only need to know 1D and 2D lists.

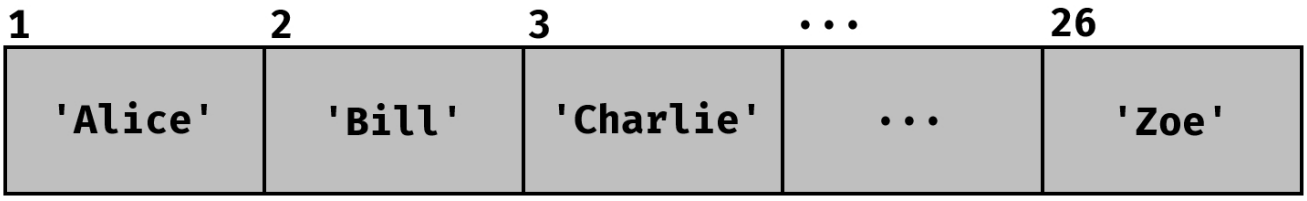
3.1.3. Accessing List Elements

Now we know how to create lists, but how do we access the information inside the list? Just like how we created the list, we can use the square bracket operator (`[]`) to access the list elements. We also call this the *index operator*. But first, we have to become comfortable with how list *indices* work in Python (and other languages as well).

Let's look at the example list we covered before:

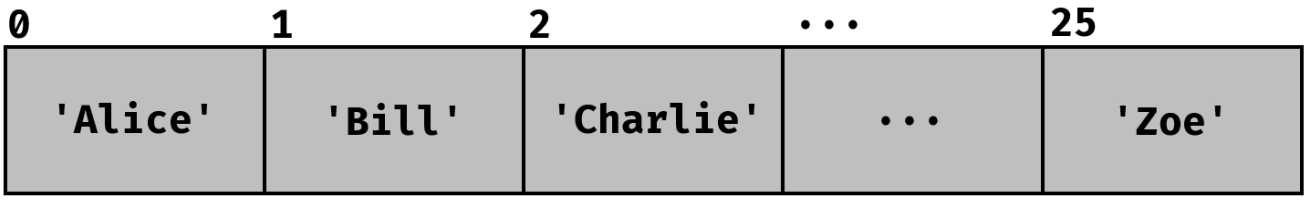


In the non-programming world, if we were to give indices to the elements in `name_list`, we would do it as follows:



`'Alice'` would be at index `1`, `'Bill'` would be at index `2`, and so on.

However, Python (and most other languages) uses a zero-based indexing system, which means the start index is 0, like the following:



Notice that for 26 elements, the index starts from `0` and ends at `25`. As such, for any list with `n` elements, the index will go from `0` to `n - 1`.

> **NOTE** `n` is the *length* of the list

So if we want to access the elements in `name_list`, we can do the following:

```
# Suppose name_list is already initialized and filled with values
name_list[0]   # -> 'Alice'
name_list[1]   # -> 'Bill'
name_list[25]  # -> 'Zoe'
```

Out of Range Indices

Now we know that the index system of a list with `n` elements will go from `0` to `n - 1`. What would

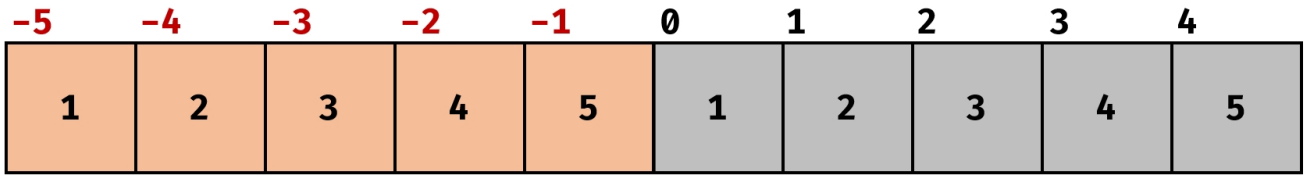
happen if we tried to access an index that is less than `0` or greater than `n - 1`? Let's try it out in Python shell.

Out of Range Index (Python Shell)

```
>>> numbers = [1, 2, 3, 4, 5]  # index: 0 ~ 4
>>> numbers[-1]
5
>>> numbers[-5]
1
>>> numbers[-6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> numbers[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

The description and name of the error do not matter, but the results seem strange, doesn't it? Why do some negative numbers work while others cause errors? If negative numbers work, why don't numbers greater than `4` work?

Python is unique in that it allows some range of negative index numbers to allow accessing of elements in reverse order. In fact, for a list with `n` elements, the index does not start from `0`, but from `-n` to `n - 1`. Therefore, the list `numbers` has the following indexing system:



You can also think of negative indices as the following:

```
1    # a has 5 elements
2    a = [1, 2, 3, 4, 5]
3    n = 5  # length of a
4
5    # Negative index can be offset by the list's length
6    a[-5]  # equivalent to a[-5 + n] = a[0]
7    a[-1]  # equivalent to a[-1 + n] = a[4]
```

Now, try comparing this figure with the shell output above. The reason why `numbers[-6]` and `numbers[5]` caused errors is because they were out of the valid index range (`-5` to `4`). For any list with `n` elements, if you try to access an index that is out of range (from `-n` to `n - 1`), Python will throw an error.

3.1.4. List Operations, Functions, and Methods

To use lists more effectively, we need to know the different operators, functions, and methods we can use with lists.

3.1.4.1. List Operators

Operator	Name	Example	Description
`[]`	Index Operator	`lst[x]`	Accesses element at specified index.
`+`	`+` Operator	`lst_a + lst_b`	Concatenates the second list to the end of the first list.
`*`	`*` Operator	`lst * n`	Repeats the list by the specified number of times.
`==`	Equivalence Operator	`lst_a == lst_b`	Checks if the two lists are equal.
`in`	Membership Operator	`el in lst`	Checks if the list contains the specified element.
`:`	Slice Operator	`lst[x:y]`	*Slices* the list from index `x` (inclusive) to index `y` (exclusive).
`del`	`del` Operator	`del el`	Deletes specified element(s) from the list.

`+` Operator Further Example

```
1    a = [1, 2, 3]
2    b = [4, 5, 6]
3    c = a + b
4    print(c)
5
6    d = ['a', 'b', 'c']
7    e = a + d
8    print(e)
```

Output

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 'a', 'b', 'c']
```

For any expression in the form `a + b`, where `a` and `b` are lists, `b` will be concatenated to the end of `a`.

In line 3, since `b` (= `[4, 5, 6]`) is concatenated to the end of `a` (= `[1, 2, 3]`), the list `[1, 2, 3, 4, 5, 6]` is assigned to the variable `c`.

In line 7, `d` (= `['a', 'b', 'c']`) is concatenated to the end of `a` (= `[1, 2, 3]`), so the list `[1, 2, 3, 'a', 'b', 'c']` is assigned to the variable `e`.

`*` Operator Further Example

```
1    a = [1, 2, 3]
2    b = a * 3
3    print(b)
```

Output

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

For any expression in the form `a * b`, where `a` is a list and `b` is an integer, `a` will be repeated `b` times.

In line 7, since `a` (= `[1, 2, 3]`) is repeated `b` (= `3`) times, the list `[1, 2, 3, 1, 2, 3, 1, 2, 3]` is assigned to the variable `b`.

We can see that `a * b` is the same as adding `a` to each other `b` times. For example, in line 2, `b = a * 3` would be equivalent to writing `b = a + a + a`.

Equivalence Operator Further Example

```
1    a = [1, 2, 3]
2    b = [1, 2, 3]
3    c = [1, 2, 3, 4]
4
5    print(a == b)
6    print(a == c)
```

Output

```
True
False
```

For any expression in the form `a == b`, where `a` and `b` are lists, it will evaluate to `True` if for all index `i`, `a[i] == b[i]` is `True`. In other words, the elements of `a` and `b` must be equal and in the same order.

In line 5, we check if the contents of `a` (= `[1, 2, 3]`) and `b` (= `[1, 2, 3]`) are equal and in the same order. Since they are, `True` is printed.

In line 6, we check if the contents of `a` (= `[1, 2, 3]`) and `c` (= `[1, 2, 3, 4]`) are equal and in the same order. Since they are not, `False` is printed (`a[3]` is invalid while `c[3]` is `4`, so `a[i] == c[i]` is `False` when `i` is `4`).

Similarly, we can use `!=` to check if the two lists are *not equal*.

Membership Operator Further Example

```
1    a = [1, 2, 3]
2    b = 1 in a
3    print(b)
4
5    c, d = ['a', 'b', 'c'], 'd'
6    e = d in c
7    f = d not in c
8    print(e, f)
```

Output

```
True
False True
```

For any expression in the form `a in b`, where `a` is any value and `b` is a list, it will return the boolean value `True` if `a` is present in `b`, and `False` if otherwise.

In line 2, since it is checking if the integer value `1` is present in `a` (= `[1, 2, 3]`), the boolean value `True` is assigned to the variable `b`.

In line 6, since it is checking if `d` (= `'d'`) is present in `c` (= `['a', 'b', 'c']`), the boolean value `False` is assigned to the variable `e`. Line 6 shows that we can use variables with membership operators.

In line 7, it shows that we can use the `not` operator with `in` to check if the element is not present in the list. Since `d` (= `'d'`) is not present in `c` (= `['a', 'b', 'c']`), the boolean value `True` is assigned to the variable `f`.

Slice Operator Further Example

```
1    a = [1, 2, 3, 4, 5, 6]  # a has length of 6
2    print(a[2:5])
3    print(a[0:6])
4    print(a[:])
5
6    # Can also use negative indices
7    print(a[-1:1])
8    print(a[-6:6])
9
10   # Index can also be out of valid range
11   print(a[4:100])
```

Output

```
[3, 4, 5]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
[]
[1, 2, 3, 4, 5, 6]
[5, 6]
```

For any expression in the form `a[x:y]`, where `a` is a list and `x`, `y` are integers, it will return a list containing elements of `a` from index `x` (inclusive) to index `y` (exclusive). This means element at index `y` will not be included.

In line 2, the list `a` (= `[1, 2, 3, 4, 5, 6]`) is sliced from index `2` to index `5`. Element at index `2` is `3`, and element at index `5` is `6`. So the resulting sliced list is `[3, 4, 5]`, excluding `6`.

In line 3, the list `a` is sliced from index `0` to index `6`. If you recall, when accessing elements, the valid index range for a list with `6` elements is `-6` to `5`. However, since the element at second index is excluded when slicing, we can use the index `6` to include the last element of the list `a`. Therefore, for any list, say `lst`, with `n` elements, `lst[0:n]` is equal to `lst` (i.e. `lst[0:n] == lst` evaluates to `True`).

In line 4, no indices are specified. If the first index is not specified, it is defaulted to `0`, and if the second index is not specified, it is defaulted to `n` (length of list). In the case of line 4, since both indices are not specified, `a[:]` is defaulted to `a[0:6]`, thus printing the whole list. Similarly, `a[:4]` would be equivalent to `a[0:4]`, and `a[4:]` would be equivalent to `a[4:6]`.

Lines 7 and 8 show that we can use negative indices when slicing. Line 7 is equivalent to writing `print(a[5:1])`. Since the start index is greater than the end index, it returns an empty list (`[]`). Line 8 is equivalent to writing `print(a[0:6])`, which is the same as line 3.

Line 11 shows that we can use any integer when slicing. The indices do not have to be in the valid range. Suppose a list has a length of `n`. If the number provided is greater than `n`, it will default to `n`. If the number provided is less than `-n`, it will default to `-n` (or `0`). Therefore, line 11 is equivalent to writing `print(a[4:6])`.

Updating Elements

To update list elements, we use the combination of assignment and index operators.

Update Example

```
1    lst = [1, 2, 6, 4, 5]
2    lst[2] = 3
3    print(lst)
```

Output

```
[1, 2, 3, 4, 5]
```

In the above example, line 2 is where the update is happening. If we look at the line closely, we can see that we are assigning the integer value `3` to the 2nd index of the list `lst`. Since indexing in Python is zero-based, the third element `6` is changed to `3`.

We can also use the slice operator to update a range of elements.

Bulk Update Example

```
 1    a = [1, 2, 3, 4, 5, 6]
 2    a[2:4] = [10, 11]
 3    print(a)
 4
 5    a = [1, 2, 3, 4, 5, 6]
 6    a[2:4] = [10, 11, 12]
 7    print(a)
 8
 9    a = [1, 2, 3, 4, 5, 6]
10    a[2:4] = [10]
11    print(a)
```

Output

```
[1, 2, 10, 11, 5, 6]
[1, 2, 10, 11, 12, 5, 6]
[1, 2, 10, 5, 6]
```

In lines 2, 6, and 10, we are updating a range of elements of list `a`, specifically from index `2` to index `4` (exclusive). In other words, we are updating `[3, 4]` of `a` (= `[1, 2, 3, 4, 5, 6]`).

In line 2, we are updating `[3, 4]` with `[10, 11]`. Hence, `a` is changed from `[1, 2, 3, 4, 5, 6]` to `[1, 2, 10, 11, 5, 6]`.

Lines 6 and 10 show that the numbers of elements to update can be different. For example, `a[2:4]` corresponds to 2 elements, but in line 6, we are assigning a list of 3 elements. As a result, `[3, 4]` of `a` is changed to `[10, 11, 12]`, ultimately updating `a` from `[1, 2, 3, 4, 5, 6]` to `[1, 2, 10, 11, 12, 5, 6]`.

Similarly, in line 10, `a` is changed from `[1, 2, 3, 4, 5, 6]` to `[1, 2, 10, 5, 6]`.

Deleting Elements

`del` Operator Further Example

```
1    a = [1, 2, 3, 4, 5, 6]
2    del a[1]
3    print(a)
4
5    # Can also use splice operator to delete a range of elements
6    a = [1, 2, 3, 4, 5, 6]
7    del a[2:5]
8    print(a)
```

Output

```
[1, 3, 4, 5, 6]
[1, 2, 6]
```

The `del` operator is a reserved keyword, shown in a chart earlier in Chapter 2. It deletes the specified element(s).

In line 2, we are telling Python to delete the element at index `1` of the list `a`. As a result, `a` is changed from `[1, 2, 3, 4, 5, 6]` to `[1, 3, 4, 5, 6]`.

In line 7, we are telling Python to delete the elements in the range of index `2` to index `5` (exclusive). As a result, `a` is changed from `[1, 2, 3, 4, 5, 6]` to `[1, 2, 6]`.

3.1.4.2. Lists and Functions

What is a function? We will cover this in more detail later, but simply put, it is something that takes an input and converts it to an output. For example, if the function `f(x) = x + 3` takes `x = 6` as input, the output would be `9`. In computer programming, we also refer to the input as a *parameter* and the output as the *return value*.

Some examples of a function we have encountered so far are `print()`, `type()`, and `int()`. `print()` takes any value as an input and prints it as output. `type()` takes any value as the input and prints its variable type as output. `int()` takes any value and outputs the integer form of that value (if applicable).

These are some functions that can take lists as a parameter:

Function	Example	Description
`len()`	`len(a)`	Returns the length of list `a`
`max()`	`max(a)`	Returns the maximum element of list `a`
`min()`	`min(a)`	Returns the minimum element of list `a`
`sorted()`	`sorted(a)`	Returns the sorted list of list `a`
`sum()`	`sum(a)`	Returns the sum of all elements of list `a`

Examples of Lists and Functions

```
1    a = [4, 2, 5, 1, 3, 6]
2    b = ['c', 'a', 'A', 'C', 'b', 'B']
3
4    print('len(a) =', len(a))
5    print('len(b) =', len(b), '\n')
6
7    print('max(a) =', max(a))
8    print('max(b) =', max(b), '\n')
9
10   print('min(a) =', min(a))
11   print('min(b) =', min(b), '\n')
12
13   print('sorted(a) =', sorted(a))
14   print('sorted(b) =', sorted(b), '\n')
15
16   print('sum(a) =', sum(a))
```

Output

```
len(a) = 6
len(b) = 6

max(a) = 6
max(b) = c

min(a) = 1
min(b) = A

sorted(a) = [1, 2, 3, 4, 5, 6]
sorted(b) = ['A', 'B', 'C', 'a', 'b', 'c']

sum(a) = 21
```

`len(a)`, where `a` is a list, will return the length of `a` (i.e. the number of elements in `a`). Since both `a` and `b` have 6 elements, lines 4 and 5 print `6`.

`max(a)`, where `a` is a list, will return the maximum element of `a`. The formal definition of a maximum element `M` is as follows:

> for all index `i` of list `a`, if `M >= a[i]` is `True`, then `M` is the maximum element of `a`

Basically, the maximum element will always be greater than or equal to (`>=`) all the other elements of the list. This may seem obvious for numbers, but it can be helpful when thinking of strings or other objects in the future. In line 8, since the ASCII value for lowercase letters is greater than that of uppercase letters, `'c'` is the maximum element of list `b`.

`min(a)`, where `a` is a list, will return the minimum element of `a`. Similarly, the formal definition of a minimum element `m` is as follows:

> for all index `i` of list `a`, if `m ⇐ a[i]` is `True`, then `m` is the minimum element of `a`

In line 11, since the ASCII value for uppercase letters is less than that of lowercase letters, `'A'` is the minimum element of list `b`.

`sorted(a)`, where `a` is a list, will return `a` sorted in increasing order. In other words, if index `x` is less than `y`, `a[x]` will be less than or equal to `a[y]` (i.e. `a[x] ⇐ a[y]` is `True`).
In line 14, since uppercase letters have lower ASCII values than lowercase letters, `sorted(b)` is `['A', 'B', 'C', 'a', 'b', 'c']`.

Notice that there is no example for `sum(b)`. This is because `sum()` only works if the list contains only integers.

3.1.4.3. List Methods

What is a method? A method is a function that is associated with an object. So what is the difference between a function and a method?

Suppose we have a list `lst`, a function `f()`, and a method `m()`.

To use the function `f()`, we would write `f(lst)`. The function takes `lst` as a parameter, or input.

To use the method `m()`, we would write `lst.m()` because the method `m()` is associated with the list `lst` object. The method `m()` does not take `lst` as a parameter.

We will cover this in more depth later. For now, you can think of it as something similar to a function.

These are some of Python's built-in list methods:

Method	Example	Description
`append()`		
`extend()`		
`insert()`		
`clear()`		
`index()`		
`count()`		
`sort()`		
`reverse()`		
`remove()`		
`pop()`		

Deleting Elements

3.2. Strings

3.2.1. Similarities with List

3.2.2. String Operations, Functions, and Methods

3.3. If Statements

3.3.1. Conditional Execution

3.3.2. Alternative Execution

3.3.3. Nested If Statements

3.3.4. Comparison to Other Languages

3.3.5. Short Circuiting of Logical Expressions

3.4. For and While Loops

3.4.1. For Loops

3.4.2. While Loops

Infinite Loops

3.4.3. Use with Lists and Strings

3.5. List Comprehension

3.5.1. Structure

3.5.2. Conditional Statements

3.5.3. Nested List Comprehension

3.5.3.1. List of Lists

3.5.3.2. Flattened Lists