BRAC UNIVERSITY

Inspiring Excellence

Course Code: CSE110

Course Title: Programming Language I

Name: Iftekhar Hossain Rahi
Student ID: 22201168
Section: 23

For many of you, it is your very first programming course. Here we will give you a brief idea about how to solve problems by writing programs in Python.

Data types, Values and Variables

Data types

The data type is mainly the category of the data. There are basically 5 groups of main data types but there are various other types available in python.

1. Numeric type:
    a) Integer (int): Positive or negative whole numbers (without a fractional part).
    For example: 10,1000000,-3
    b) Floating-point (float): Any real numbers with "decimal" points or floating-point representation.
    For example: -3.1234, 100.3458998

2. Boolean type (bool): True and False.

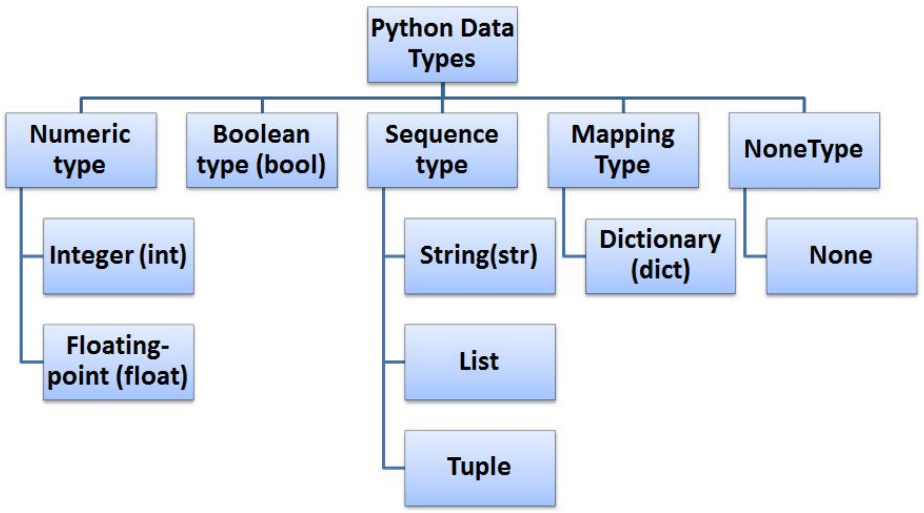
3. Sequence type: A sequence is an ordered collection of similar or different data types.

a) String(str): It is a sequence of ordered characters (alphabets-lower case, upper case, numeric values, special symbols) represented with quotation marks (single('), double(''), triple(' ' ', '' '' '')).
For example: using single quotes ('CSE110'), double quotes ("CSE110") or triple quotes ('''CSE110''' or """CSE110"""), "Hello CSE110 students".
Note: Details will be discussed later.

b) List: It is an ordered collection of elements where the elements are separated with a comma (,) and enclosed within square brackets []. The list can have elements with more than ode data types.
For example: list with only integers [1, 2, 3] and a list with mixed data types [110, "CSE110", 12.4550, [], None]. Here, 110 is an integer, "CSE110" is a string, 12.4550 is a floating-point number, [] is a list, None is NoneType.
Note: Details will be discussed later.

c) Tuple (Note: This topic will be discussed later.)

4. Mapping Type: Dictionary (dict) (Note: will be discussed later.)
5.NoneType(None): It refers to a null value or no value at all. It's a special data type with a single value, None.
Note: Details will be discussed later.



Data type checking

Mainly for debugging (code correcting) purpose type() function is used. It returns the type of argument(object) passed as a parameter.

Example:

Code	**Output**
`type("Hello CSE110")`	`str`
`type(110)`	`int`
`type(123.456)`	`float`
`type(True)`	`bool`
`type(None)`	`NoneType`

`print()` function

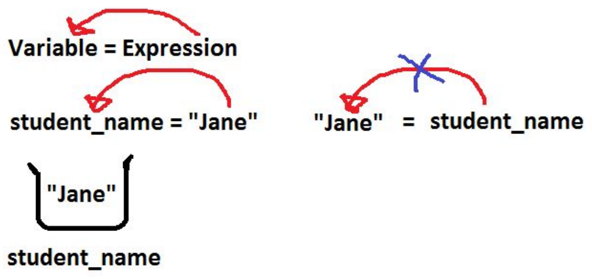
`the print()` function is used for printing the value of an expression written inside the parentheses (pair of an opening and closing first brackets). We shall discuss more about this after you have learned what is a function in python. In a command-line interpreter, you may not need the `print()` function, but when we write a set of code and want to see any text output on the screen, we use the `print()` function. For example:

Code	**Output**
`print("Hello CSE110 class")`	`Hello CSE110 class`

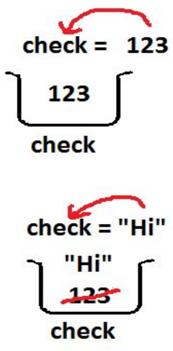
`print(123)`	`123`
`print(123.567)`	`123.567`
`print(type(123.456))`	`<class 'float'>`

Variable

Variables are containers for storing data values.



Here, student_name is a variable and it is holding or referring to the value “Jane”. Values are always stored from the right side of the assignment(=) to the variable on the left side. Unlike other languages, in python, we do not need to declare data types directly. The data type is decided by the interpreter during the run-time.



one variable can be assigned values of different data types. Here, in this example, first 123 was put in the container or variable named check. Then we replaced the value of the container from 123 to “Hi” which is a string. For example:

Code	Output

`check = 123`
`print(check)`
`print(type(check))`
`print("======")`
	`123`
	`<class 'int'>`
	`======`
	`Hi`

```
check = "Hi"
print(check)
print(type(check))
```

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

Variable naming conventions

While naming the variables we need to follow certain rules and restrictions. These are called “Variable naming conventions”. By following these our codes become more understandable by the Human Readers (us). Few significant rules have been mentioned below.

Variable names-

- can have letters (A-Z and a-z), digits(0-9), and underscores (_).
- should maintain snake_casing. That means each word should be separated by underscores(_)
- cannot begin with digits
- cannot have whitespace and special signs (e.g.: +, -, !, @, $, #, %.)
- are case sensitive meaning ABC, Abc, abc are three different variables.
- have to be meaningful with the data stored in it.
- cannot be too lengthy and too general, need to have a balance
- should not be written with single alphabets. Minimum length should be 3.
- cannot be a reserved keyword for Python. There are a total of 35 keywords.

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

Detailed variable naming conventions can be found in the following
https://www.python.org/dev/peps/pep-0008/#prescriptive-naming-conventions

Basic operations and precedence

Now you know what data types are. A programming language uses ‘operations’ to manipulate the data stored in variables to achieve the desired results. Basic operations in Python can be divided into two parts: Unary (meaning it is done with one variable) and Binary (meaning it is done using two variables or one variable and a single value of data).
Note: You cannot do any operation with None type.

Let’s explore each type of operation and understand what they do.

Unary operations

1. Unary + (plus): We use unary + (plus) operation by adding a ‘+’ before a variable or data. It does not change the data. (Works with int, float, complex, and boolean. For booleans, True and False will be valued as 1 and 0 respectively.) For example:

Code	Output

`x = 5`
`print(x)`
`print(+x)`
	`5`
	`5`

2. Unary - (minus): We use unary - (minus) operation by adding a ‘-’ before a variable or data. It produces the negative value of the input (equivalent to the multiplication with -1). (Works with int, float, complex, and boolean. For booleans, True and False will be valued as 1 and 0 respectively.) For example:

Code	Output

`x = 5`
`print(x)`
`print(-x)`
	`5`
	`-5`

3. Unary ~ (invert): We use unary - (invert) operation by adding a ‘~’ before a variable or data. It produces a bitwise inverse of a given data. Simply, for any data x, a bitwise inverse is defined in python as -(x+1). (Works with int, and boolean. For booleans, True and False will be valued as 1 and 0 respectively.) For example:

Code	Output

`x = 5`
`print(x)`
`print(~x)`
	5
	-6

Binary operation:

Operators are symbols that represent any kind of computation such as addition, subtraction, and etc. The values or the variables the operator works on are called Operands.

1 + 2

here, 1 and 2 are operands, and + is the operator computing addition.

1. Arithmetic operation
    a. + (addition)
    b. - (subtraction)
    c. * (multiplication)

Code	Output
`print(2+3)`	5
`print(2-30)`	-28
`print(2*30)`	60
`print(2.456789*30)`	73.70367

    d. / (division)

```
Division of numbers(float, int) yields a float
```

Code	Output

`print(4/2)`
`print(9/2)`
	2.0
	4.5

`print(-5/2)`	-2.5
`print(30/4)`	7.5

    e. // (floor division)

```
Floor division of integers results in an integer.
If one of the operands is float, then the floor division
```

results in a float.

Code	Output
`print(30//4)`	7 (here, from the previous example, we can see, 30 divided by 4 yields 7.5. But while using the floor division (//), it basically discards the part after the decimal point and returns a whole number by flooring the value to the nearest integer number. Or we can say that it returns the quotient value after dividing)

`print(10.5//2.5)`
`print(10.5//2)`
`print(10//2.5)`
	4.0
	5.0
	4.0
	
	(If one of the operands is float, then the floor division results in a float.)

`print(-5//2)`	-3 Here, -5 divide by 2 gives -2.5 . Since we have used floor divide, it will take it to the nearest (lower) integer, which is -3.

f. % (modulus)

(0) For positive values:

`z = x%y`

Here, x is the dividend, y is the divisor and z is the remainder.

Code	Output
`print(30%4)`	2 (here, the percentage sign is called "modulus operator" and it basically returns the remainder by dividing the first number with the second number. 30 divide by 4 yields quotient,4 and remainder 2. So the final output is 2)

(1) For negative values:

First option (For negative values) :

z = x - y*(x//y)

x%y, then, the modulo operator always yields a result with the same sign as its second operand(the divisor) or zero

Code	Output

`print(-5%2)`	1
	
	Here, x = -5, y = 2
	
	z = -5 - 2*(-5//2)
	  = - 5 - 2*(-3)
	   = - 5 +6
	   = 1

`print(-5.5%2)`	0.5
	
	Here, x = -5.5 , y = 2
	
	z = -5.5 - 2*(-5.5//2)
	  = - 5.5 - 2*(-3)
	   = - 5.5 +6
	   = 0.5

`print(5%-2)`	-1
	
	Here, x = 5, y = -2
	
	z = 5 - (-2)*(5//-2)
	  = 5 - (-2)*(-3)
	  = 5 - (6)
	  = -1

`print(5.5%-2)`	Here, x = 5.5, y = -2
	
	z = 5.5 - (-2)*(5.5//-2)
	  = 5.5 - (-2)*(-3.0)
	  = 5.5 - (6.0)
	  = - 0.5

Second option (For negative values) :

x%y = (a+b)mod y = [(a mod y)+(b mod y)]mod y

Here, the x, dividend is broken into two parts such that, one part is always positive between a and b.

Code	Output
`print(-5%2)`	1 Here, -5%2 = (3-8)%2 = [3%2 + (-8%2)]%2 = (1+0)%2=1
`print(-5.5%2)`	0.5 Here, -5.5%2 = (2.5-8)%2 = [2.5%2 + (-8%2)]%2 = (0.5+0)%2 =0.5
`print(5%-2)`	-1 Here, 5%-2 = (4+1)%-2 = [(4%-2) + (1%-2)]%2 = [0+ -1]%2 = -1
`print(5.5%-2)`	-0.5 Here, 5.5%-2 = (4+1.5)%-2 = [(4%-2) + (1.5%-2)]%-2 = [0+ -1.5]%-2 = (-1.5)%-2= (-2+0.5)%-2 =[(-2%-2) + 0.5%-2]= 0+ (-0.5) = -0.5

g. ** (Exponentiation)

Basically it's the power operator (X**Y)= x^Y

Code	Output
`print(2**4)`	16
`print(-3**2)`	-9 (Here, - $3^2 = -9$)
`print((-3)**2)`	9 (Here, $(-3)^2 = 9$)
`print(4**-2)`	0.0625 Here, $4^{-2} = \frac{1}{4^2} = \frac{1}{16}$

Details can be found in the following link:
https://docs.python.org/3/reference/expressions.html#binary-arithmetic-operations

1. Assignment operator
    a. = (assign): It is used for putting value from the right side of the equal sign(=) to a variable on the left side of the equal sign(=).
    For example: number = 123.

    b. Compound Assignment Operators:
        i. += (add and assign)
        ii. -= (subtract and assign)
        iii. *= (multiply and assign)
        iv. /= (divide and assign)
        v. %= (modulus and assign)
        vi. **= (exponent and assign)
        vii. //= (floor division and assign)

Compound Assignment Operators	Example	
	Short-form	full form
`+=`	`a += 7`	`a = a + 7`
`-=`	`b -= 2`	`b = b - 2`
`*=`	`c *= 3`	`c = c * 3`
`/=`	`d /= 9`	`d = d / 9`
`%=`	`e %= 2`	`e = e % 2`
`**=`	`f **= 4`	`f = f ** 4`
`//=`	`g //= 11`	`g = g // 11`

2. Logical operator
    a. and (logical AND)
    b. or (logical OR)
    c. not (Logical NOT)
    Note: Details will be discussed in the branching chapter.

3. Comparison or Relational operator
    a. == (equal)
    b. != (not equal)

c. > (greater than)
d. < (less than)
e. >= (greater than or equal)
f. <= (less than or equal)
Note: Details will be discussed in the branching chapter.

4. Membership Operator
    a. in: Returns True if the first value is in the second. Otherwise, returns False.
    b. not in: Returns True if the first value is not in the second. Otherwise, returns False.

Example:

Code	**Output**
`print(4 in [1, 2, 3, 4, 5])`	`True`
`print("p" in "Hello")`	`False`
`print(100 not in [1, 2, 3, 4, 5])`	`True`
`print("p" not in "Hello")`	`True`

5. Identity Operators
   Identity operators check whether two values are identical or not.

    a. is: Returns True if the first value is identical or the same as the second value. Otherwise, returns False.
    b. is not: Returns False if the first value is identical or the same as the second value. Otherwise, returns True.

Example:

Code	**Output**
`print("123" is 123)`	`False`
`print("123" is "123")`	`True`
`print(5.123 is 5.1234)`	`False`
`print("123" is not 123.00)`	`True`
`print("123" is not "123")`	`False`

7. Bitwise Operators [Note: will be covered in future courses]

a. & (Bitwise and)
b. | (Bitwise or)
c. ^ (Bitwise xor)
d. ~ (Bitwise 1's complement)
e. << (Bitwise left-shift)
f. >> (Bitwise right-shift)

Compound expression:

When we write an expression with two or more operations, it is called a compound expression. For example, we may write `7+9*3` where we have both addition and multiplication operations in a single expression. Determining the result of a compound expression is a little tricky. We need to think about what operation will be executed first. To determine that the computer follows Operator precedence, a set of rules that dictates the computer should be done first. For our example, `7+9*3` the multiplication operation will have higher precedence and will be executed first. So the program will first calculate `9*3` which results in 27. Then it will calculate 7+27, which will result in 34.

Try the following examples and see what results in it shows:

5*3+2-1*2

1+7/7*7

Operator precedence:

In the tale below precedence has been shown in descending order. Highest precedence at the top, lowest at the bottom. Operators in the same precedence are evaluated from left to right.

Precedence	**Operator**	**Meaning**
Highest	`()`	Parentheses (grouping)
	`f(args...)`	Function call
	`x[index:index]`	Slicing
	`x[index]`	Subscription
	`x.attribute`	Attribute reference
	`**`	Exponentiation
	`~x`	Bitwise not
	`+x, -x`	Positive, negative

	*, /, %, //	Multiplication, division, modulus, floor division
	+, -	Addition, subtraction
	<<, >>	Bitwise shifts
	&	Bitwise AND
	^	Bitwise XOR
	|	Bitwise OR
	in, not in, is, is not, <, <=, >, >=, <>, !=, ==	Comparisons, membership, identity
	not *x*	Boolean NOT
	and	Boolean AND
	or	Boolean OR
Lowest	lambda	Lambda expression

Example:	
Code	**Explanation and output**

print(True or False and True)	here, "and" has precedence over "or"
	
	print(True or (False and True))
	print(True or (False))
	print(True or False)
	print(True)
	
	Output: True

print((True or False) and True)	here, parenthesis "() " has precedence over both "or" and "and"
	
	print((True or False) and True)
	print((True) and True)
	print(True and True)
	
	Output: True

print(9 - 5 // 2 * 13 + 2 ** 2)	here, Exponentiation (**) has the highest precedence. So, 2**2 will be executed first
	
	print(9 - 5 // 2 * 13 + 2 ** 2)
	
	in the next step, floor division(//), and multiplication(*) has the same precedence if the precedence is the same then, we need to execute from left to right. So, 5 // 2
	
	print(9 - 5 // 2 * 13 + 4)
	print(9 - 2 * 13 + 4)
	
	then, * has the highest precedence. So 2 * 13 executes
	
	print(9 - 2 * 13 + 4)
	print(9 - 26 + 4)
	
	Here,- and + has the same precedence, so again left to right.
	
	print(9 - 26 + 4)
	print(-17 + 4)
	print(-13)
	
	Output: -13

print(1.5 * 10 // 2 + 8)	here, * and // has the same precedence, so execute from left to right. Execute 1.5 * 10
	
	print(1.5 * 10 // 2 + 8)
	print(15.0 // 2 + 8)
	
	Then execute 15.0 // 2
	
	print(15.0 // 2 + 8)
	print(7.0 + 8)
	print(7.0 + 8)
	print(15.0)
	
	Output: 15.0

`print(-3 ** 2 * 2.0)`	here, `**` has precedence over `*`.
	
	`print(-3 ** 2 * 2.0)`
	`print(-9 * 2.0)`
	`print(-9 * 2.0)`
	`print(-18.0)`
	
	`Output: -18.0`

`print(-3 ** ( 2 * 2.0))`	here, `()` has precedence over `**`.
	
	`print(-3 ** ( 2 * 2.0))`
	`print(-3 ** ( 4.0))`
	`print(-3 ** 4.0)`
	`print(-81.0)`
	
	`Output: -81.0`

Type Conversion:

Data types can be modified in two ways: Implicit (when the compiler changes the type of a data for you) and Explicit (when you change the type of a data using type changing functions, it is called *"Type-casting"*)

Note: What is the compiler will be explained later. Assume it to be like a teacher who checks your code for any kind of mistakes and shows the problems(errors) and sometimes does type conversion for you according to the computation need of a statement.

1. Implicit Type Conversion:
   If any of the operands are floating-point, then arithmetic operation yields a floating-point value. If the result was integer instead of floating-point, then removal of the fractional part would lead to the loss of information.

Operation	**Result**
`5.0 -3`	`2.0`
`4.0/5`	`0.8`
`4/5.0`	`0.8`
`4*3.0`	`12.0`

`5//2.0`	`2.0`
`7.0%2`	`1.0`

2. Explicit Type Conversion:
   Conversion among different data types are possible by using type conversion functions, though there are few restrictions. Python has several functions for this purpose among them these 3 are most used:
   a. str() : constructs a string from various data types such as strings, integer numbers and float-point numbers. For example:

Code	**Output**

`print(str(1))`
`print(type(str(1)))`
`print("=============")`
 	
`print(str(1.555))`
`print(type(str(1)))`
	`1`
	`<class 'str'>`
	`=============`
	`1.555`
	`<class 'str'>`

   b. int(): constructs an integer number from various data types such as strings (the input string has to consist of numbers without any decimal points, basically whole numbers), and float-point numbers (by rounding up to a whole number, basically it truncates the decimal part). For example:

Code	**Output**

`print(int("12"))`
`print(type(int("12")))`
`print("=============")`
 	
`print(int(12.34))`
`print(type(int(12.34)))`
	`12`
	`<class 'int'>`
	`=============`
	`12`
	`<class 'int'>`

`print(int("12.34"))`	`ValueError`
`print(int("12b"))`	`ValueError`

   c. float(): constructs a floating-point number from various data types such as integer numbers, and strings (the input string has to be a whole number or a floating point number). For example:

Code	Output

`print(float(12))`
`print(type(float(12)))`
`print("=============")`
 	
`print(float("34"))`
`print(type(float("34")))`
`print("=============")`
 	
`print(float("34.56789"))`
`print(type(float("34.56789")))`
`print("=============")`
	`12.0`
	`<class 'float'>`
	`=============`
	`34.0`
	`<class 'float'>`
	`=============`
	`34.56789`
	`<class 'float'>`
	`=============`

`print(float("34.5b789"))`	`ValueError`
`print(float("34b6"))`	`ValueError`

Input

For taking input from the user directly, Python has a special built-in function, input(). It takes a String as a prompt argument and it is optional. It is used to display information or messages to the user regarding the input. For example, “Please enter a number” is a prompt. After typing the data/input in the designated inbox box provided by the IDE, the user needs to press the ENTER key, otherwise the program will be waiting for the user input indefinitely. The input() function converts it to a string and then returns the string to be assigned to the target variable.

Example: Taking name(str) and age(int) as input from user

```
name = input("Please enter your name: ")
print("Your name is " + name)
print("name data type: ", type(name))

age = input("Please enter your age: ")
print("Your age is " + age)
print("age data type: ", type(age))
```

Input	Output

`Jane`
`20`
	`Please enter your name: Jane`
	`Your name is Jane`
	`name data type: <class 'str'>`
	`Please enter your age: 20`
	`Your age is 20`
	`age data type: <class 'str'>`

Here, from the example output, we can see that age has a String data type which was integer during the initial data input phase. This implicit data conversion has taken place in the input() function. No matter what data type we give as an input, this function always converts it to a string. So in order to get a number as an input, we need to use **explicit type conversion** on the input value.

Example: Taking a floating-point as an input

```
number = float(input("Please enter a floating-point number:"))
print("You have entered:"+ str (number))
print("Data type is",type(number))
```

Input	Output
`12.75`	`Please enter a floating-point`

	`number:12.75`
	`You have entered:12.75`
	`Data type is <class 'float'>`

Example: Taking integer as a input

```
number = int(input("Please enter a number:"))
print("You have entered: "+ str (number))
print("Data type is",type(number))
```

Input	Output

`13`	`Please enter a number:13`
	`You have entered: 13`
	`Data type is <class 'int'>`

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/

Branching

Boolean expression

In boolean expression, we compare two values or statements using the comparison operator to yield a boolean value (True or False).

Comparison or Relational operator

a. == (equal): returns True if the first value is equal to the second. [single equal(=) is used for value assignment but double equal (==) is used for value equality]
b. != (not equal): returns True if the first value is not equal to the second.
c. > (greater than): returns True if the first value is greater than the second.
d. < (less than): returns True if the first value is less than the second
e. >= (greater than or equal): returns True if the first value is greater than or equal to the second.
f. <= (less than or equal): returns True if the first value is smaller than or equal to the second.

Example:

Code	Output

`print(5==5)`
`print(5==50)`
	`True`
	`False`

`print(5!=5)`
`print(5!=50)`
	`False`
	`True`

`print(100>2)`
`print(1>20)`
	`True`
	`False`

`print(1<20)`
`print(100<2)`
	`True`
	`False`

`print(1>=20)`
`print(100>=2)`
`print(100>=100)`
	`False`
	`True`
	`True`

`print(100<=2)`
`print(2<=2)`
`print(1<=100)`
	`False`
	`True`
	`True`

Logical operator

a. and (logical AND): The logical and returns True if both values are True. Otherwise, returns False.

b. or (logical OR): The logical or returns False if both values are False. Otherwise, returns True.
c. not (Logical NOT): Returns True, if False is given and vise versa.

A	B	A and B	A or B	not A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Example:

Code	Output	Explanation

`variable1 = True`
`print(not variable1))`
	False	`print(not variable1)`
		`print(not True)`
		`print(not True)`
		`print(False)`
		
		`Output: False`

`variable1 = False`
`print(not variable1)`
	True	`print(not variable1)`
		`print(not False)`
		`print(not False)`
		`print(True)`
		
		`Output: True`

`variable1 = True`
`variable2 = True`
`print(variable1 and variable2)`
	True	`print(variable1 and variable2)`
		`print(True and True)`
		`print(True)`
		
		`Output: True`

`variable1 = True`
`variable2 = False`
`print(variable1 and variable2)`
	False	`print(variable1 and variable2)`
		`print(True and False)`
		`print(True and False)`
		`print(False)`
		
		`Output: False`

`variable1 = True`
`variable2 = False`
`print(variable1 or`
	True	`print(variable1 or variable2)`
		`print(True or False)`
		`print(True or False)`

`variable2)`		`print(True)`
		
		`Output: True`

`variable1 = False`
`variable2 = False`
`print(variable1 or variable2)`
	False	`print(variable1 or variable2)`
		`print(False or False)`
		`print(False or False)`
		`print(False)`
		
		`Output: False`

Compound Boolean expression (Logical and comparison operator combined) examples:

Code	Output	Explanation

`print(not (5>1))`	False	`Here,(5>1)is True.`
		`print(not (5>1))`
		`print(not (True))`
		`print(not True)`
		`print(False)`
		
		`Output: False`

`print(not (5>500))`	True	`Here,(5>500)is False.`
		`print(not (5>500))`
		`print(not (False))`
		`print(not False)`
		`print(True)`
		
		`Output: True`

`print((5>1) and (5<20))`	True	`Here,(5>1)is True and (5<20) is True.`
		`print((5>1) and (5<20))`
		`print((True) and (True))`
		`print(True and True)`
		`print(True)`
		
		`Output: True`

`print((5>1) and (5>500))`	False	`Here,(5>1)is True and (5>500) is False.`
		`print((5>1) and (5>500))`
		`print((True) and (False))`
		`print(True and False)`
		`print(False)`
		
		`Output: False`

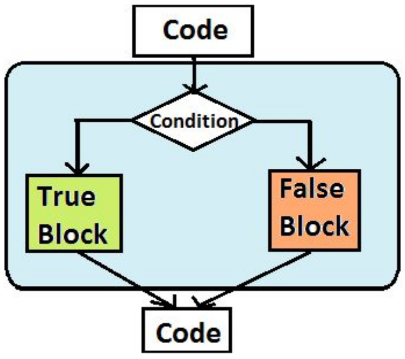
`print((5>1) or (5==50))`	True	Here, (5>1) is True and (5==50) is False.
		`print((5>1) or (5==50))`
		`print((True) or (False))`
		`print(True or False)`
		`print(True)`
		
		Output: True

`print((-6>1) or (5>500))`	False	Here, (-1>1) is False and (5>500) is False.
		`print((-6>1) or (5>500))`
		`print((False) or (False))`
		`print(False or False)`
		`print(False)`
		
		Output: False

Conditional statements

Comparison operators are basically used for writing conditional statements. They have three parts:

1. a condition yielding in either True or False.
2. a block of code is executed if the condition yields True.
3. another non-mandatory block of code is executed if the condition yields False.



Indentation:

Indentation means leading whitespace (spaces and tabs) at the beginning of a particular line of code. It is basically used for grouping a section of code. A particular section or bundle of code is called "Block". For example, in the previous flowchart, "True block" holds a section of code, which will only be executed if the condition yields True and "False block" holds a section of code, which will only be executed if the condition yields False. So, for blocking or in more general terms for paragraphing one or multiple lines, we use indentations.

Note: details about indentation is provided in the link below.
Link: https://docs.python.org/2.0/ref/indentation.html

1. **Basic conditional statements:**

Basic conditional statement code structure

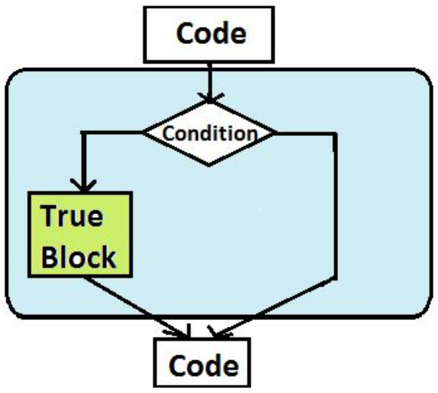
`if(condition):`
`     #codes inside True block`
`     True block`
`     #codes inside True block`
`else:`
`     #codes inside False block`
`     False block`
`     #codes inside False block`
`#codes outside False block`

Example: Code	Output

`if( 5 > 10 ):`
`     print("10 is greater")`
`else:`
`     print("5 is greater")`
	`5 is greater`

`if( 30%2==0 and 30%3==0 ):`
`     print("Even and multiple of 3")`
`else:`
`     print("not")`
	`Even and multiple of 3`

2. **Unary Selection (Omitting the else Clause)**
When the condition evaluates to True, the True block is executed and the else and False block is completely excluded.



Basic conditional statement code structure	

`if(condition):`
`     #codes inside True block`
`     True block`
`     #codes inside True block`
`#codes outside True block`	

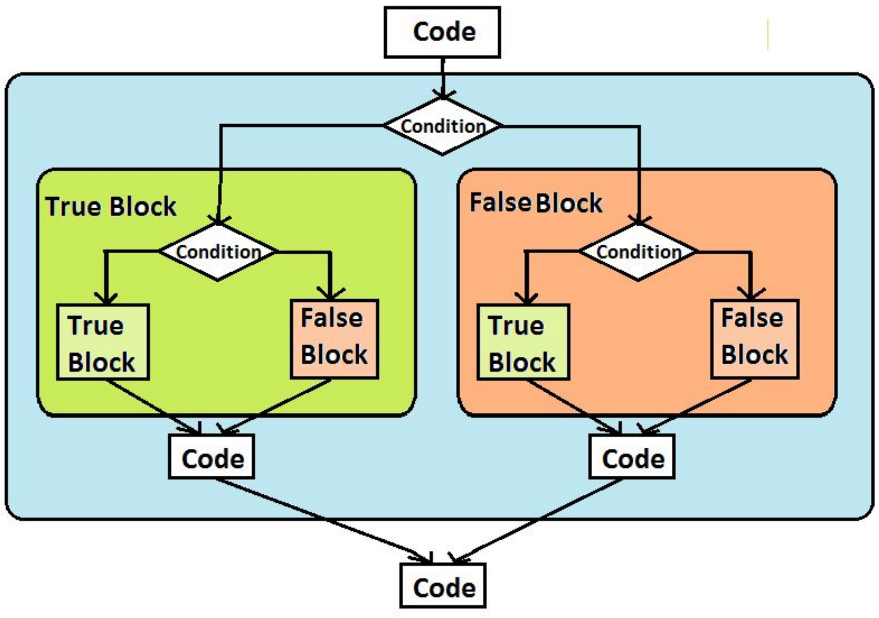
Example: Code	**Output**

`if( 5 > 10 ):`
`    print("10 is greater")`
	`NO OUTPUT IS SHOWN`

`if( 10 == 10 ):`
`    print("equal")`
	`equal`

3. **Nested conditionals**

Multiple conditional statements including unary selection statements can be nested inside one another. These are called nested conditional statements. Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. Here in the flowchart below, we can see that inside the True block and False block, there is another set of conditions with True block and False block. There True block and False block again can have conditions nested inside them.



Nested conditional statements code structure

```
if(condition):
    #codes inside True block
    if(condition):
        #codes inside True block
        if(condition):
             True block
        else:
             False block
        #codes inside True block
    else:
        False block
    #codes inside True block
else:
    #codes inside False block
    if(condition):
        True block
    else:
        False block
    #codes inside False block
#codes outside False block
```

Example: Code | **Output**

```
if 30%2==0 :
     if 30%3==0 :
         print("Even and multiple of 3")
     else:
         print("Even and not a multiple of 3")
else:
     if 30%3==0 :
         print("Odd and multiple of 3")
     else:
         print("Odd and not a multiple of 3")
```

```
Even and
multiple
of 3
```

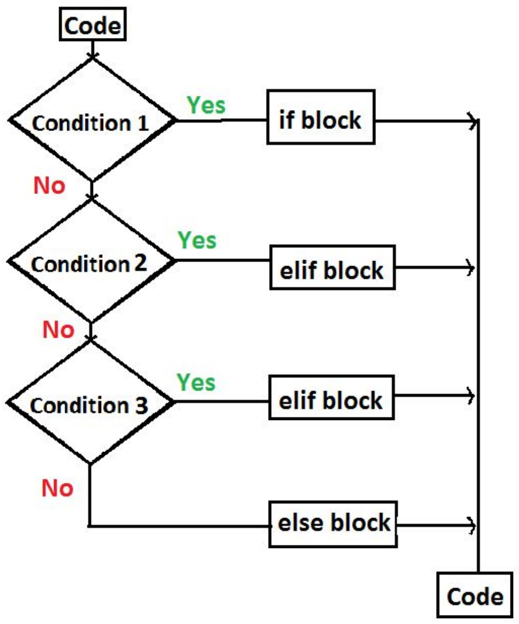
```
number = 2
if number > 0 :
     if number < 100 :
        if number % 2 == 0:
            print("Even positive number less than 100")
        else:
            print("Odd positive number less than 100")
     else:
        if number % 2 == 0:
            print("Even positive number greater than 100")
        else:
            print("Odd positive number greater than 100")
```

```
Even
positive
number
less
than 100
```

The level of nesting can be understood only with the help of Indentation. Thus it is less used by the programmers for avoiding confusion.

4. Chained or Ladder conditionals

The same checking of “nested conditionals” can be done using Chained conditionals (if….elif….elif….else). The conditions are checked from top to bottom. First the “if condition” is checked, if it yields False, then the next “elif condition” is checked. One if block, can have multiple “elif blocks” and only one “else block” at the end of the ladder, which is executed if the rest of the conditions yield False. The full form of “elif” is “else if”. Among several conditions of if...elif...else blocks, only one block is executed according to the condition.



Chained conditional statements code structure

```
if (condition):
     #codes inside if block
     if code block
     #codes inside if block
elif (condition):
     #codes inside elif block
     elif code block
     #codes inside elif block
elif (condition):
     #codes inside elif block
     elif code block
     #codes inside elif block
.
.
```

```
else:
     #codes inside else block
     else code block
     #codes inside else block
#codes outside the if-elif-else block
```

Example: Code	**Output**

`time = 4`
 	
`if time == 0:`
`     print("It is midnight")`
`elif time == 1:`
`     print("It is 1 am")`
`elif time == 2:`
`     print("It is 2 am")`
`elif time == 3:`
`     print("It is 3 am")`
`elif time == 4:`
`     print("It is 4 am")`
`else:`
`     print("5am to 12 pm")`
	`It is 4 am`

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/

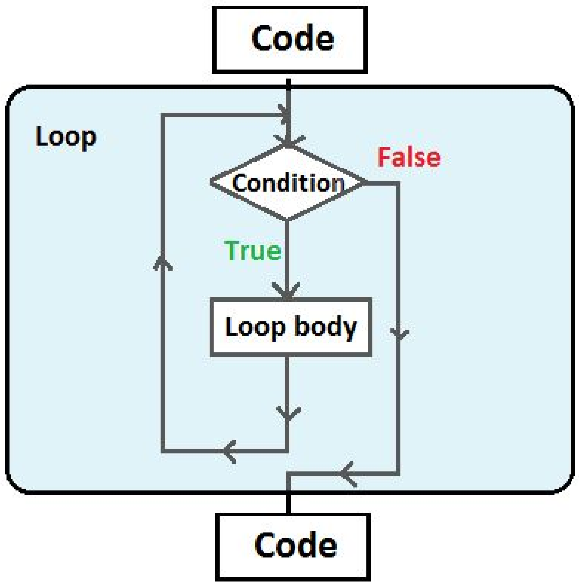
Iteration

Imagine you are asked to print “Hi” 5 times. Till now what you have learned, the most simple way to do is to print “Hi” using 5 individual print statements. What if you are asked to print “Hi” 10,000 times? then using the previous way of writing individual print statements might seem unrealistic and inefficient. Using “loops or iteration” we can solve this problem very easily. Loop gives us the advantage of doing repetitive work or we can say executing a block of statements with very few lines of code.

We have two looping constructs: while and for.

Whether to use “while” or “for” loop?

It depends on what we want to do in our program. If we want to run a block of code forever or check the condition of doing repetitive work after each iteration, then use the “while” loop. On the contrary, if we want to do repetitive work for a fixed number of times or iterate over the members/items of a sequence (string, list, range, etc.) while doing the repetitive work, then we need to use the “for” loop. Both the looping constructs have almost the same working mechanism and it is illustrated with the flowchart below.



```
print("Hi")
print("Hi")
print("Hi")
print("Hi")
print("Hi")
```

```
count = 1
while count <= 5:
        print("Hi")
        count = count + 1
```

```
for number in range(5):
    print("Hi")
```

While loops:

The basic structure of while loop
```
initialize loop controller
#codes outside while loop
while condition:# condition to terminate the loop

    #codes inside while loop
    Repetitive work
    update loop controller #(to eventually terminate the loop.)
    #(By update, we meant any arithmetic operation)
    #codes inside while loop

#codes outside while loop
```
Example: This program will print “Hi” for 100 times
```
count = 1
while count <= 100:
      print("Hi")
      count = count + 1
```

Break:

What will happen if we forget to update the loop controller?

Example: This program will print “Hi” for infinite times	Output

```
count = 1
while count <= 100:
    #codes inside while loop
    print("Hi")
    #codes inside while loop

#codes outside while loop
```
	```
	Hi
	Hi
	Hi
	Hi
	……. .
	……. .
	Hi
	```

Here the program will never terminate and it will print “Hi” for an infinite number of times (until the memory runs out). It is called “**Infinite loops**”.

Now, what can we use to stop/break this cycle or loop? For this, we have a “break” statement. It can terminate the current iteration or even the whole loop without checking the “loop terminating condition”.

Example: This program will print “Hi” for 5 times	**Output**

```
count = 1
while count <= 100:
         #codes inside while loop
         print("Hi")
         if count == 5:
               break
         count = count + 1
         #codes inside while loop

#codes outside while loop
```
	```
	Hi
	Hi
	Hi
	Hi
	Hi
	```

Example: This program will print “Hi” for 5 times (Another version of the previous program)	**Output**

```
count = 1
while True:
         #codes inside while loop
         print("Hi")
         if count == 5:
                break
         count = count + 1
         #codes inside while loop


#codes outside while loop
```
	```
	Hi
	Hi
	Hi
	Hi
	Hi
	```

Continue:

Unlike “break”, “continue” does not terminate the current loop; instead, it skips the rest of the codes of the current iteration and moves on to the next iteration.

Example: Code	**Output**

```
count = 1
while count <= 5:
    #codes inside while loop
    print("=====")
    print("Hi")
    count = count + 1

    if count == 3:
        continue
```
	```
	=====
	Hi
	Bye
	=====
	=====
	Hi
	```
	(Here, in the 3rd iteration after the checking skipped printing “bye” and the design)`=====`

```
    print("Bye")
    print("=====")
    #codes inside while loop
#codes outside while loop
```
	```
	Hi
	Bye
	=====
	=====
	Hi
	Bye
	=====
	=====
	Hi
	Bye
	=====
	```

Now, if I change the position of count = count + 1 to later in code, what problems might arise due to it? Run the code by yourself and check.

Example: Code	Output

```
count = 1
while count <= 5:
    #codes inside while loop
    print("=====")
    print("Hi")

    if count == 3:
        continue

    print("Bye")
    print("=====")
    count = count + 1
    #codes inside while loop

#codes outside while loop
```
	Check the output by running the code by yourself

Solution: During the 3rd iteration, the update of the loop controller, “counter” has been skipped by continue. So after the 3rd iteration, it prints “Hi” for an infinite time.

For loops:

The basic structure of for loop:
for <iterating_var> in <sequence/ collection>:   repetitive work or block of statements

A) Iteration of each list item using for loop:

Example: Code

```
marvel_heroes = ["Black Panther", "Captain
America", "Iron Man", "Ant-Man"]
#codes outside for loop
for hero in marvel_heroes:    # by item
     #codes inside for loop
     print(hero)
     #codes inside for loop

#codes outside for loop
```

Output

```
Black Panther
Captain America
Iron Man
Ant-Man
```

Explanation: Here, we are iterating through a list of strings named “marvel_heroes” and “hero” is the iterating variable or loop iterator which is going through all the elements of “marvel_heroes” sequentially from left to right. During each iteration, first checking is done to see whether any element or item of the sequence is left to execute and it is called the loop terminating condition. If the checking gives True, then the next item of the list is referred to the loop iterator, “hero” and the loop body is executed. Inside the loop body, the value of the loop iterator “hero” is being printed. If the checking is False, then the for loop terminates and codes outside the loop start executing sequentially.

B) Iteration of each character of String using for loop:

Example: Code

```
iron_man = "Tony Stark"
#codes outside for loop

for achar in iron_man: # by each character

     #codes inside for loop
     print(achar)
     #codes inside for loop

#codes outside for loop
```

Output

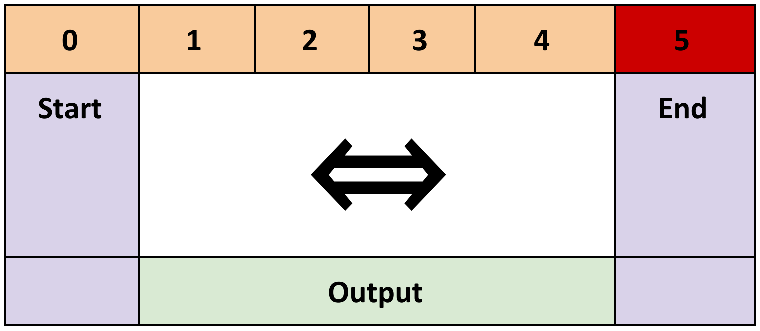
```
T
o
n
y

S
t
a
r
k
```

Explanation: Here, we are iterating through a string named “iron_man” and “achar” is the loop iterator which is going through all the characters of “iron_man” sequentially from left to right. During each iteration, first checking is done to see whether any character of the string, “iron_man” is left and it is the loop terminating condition. If the checking gives True, then the next character of the string, “iron_man” is referred to the loop iterator, “achar” and the loop body is executed. Inside the loop body, the value of the loop iterator “achar” is being printed. If the checking is False, then the for loop terminates and codes outside the loop start executing sequentially.

C) for loop using the range function:

The range function takes 3 integers (can be positive and negative) as inputs: start, end, and step size. Among these three writing end value is mandatory and when not mentioned in the function, the start has a default value of 0, and the step size has a default value of 1. Taking these inputs, the range function returns a sequence of numbers beginning from start and going up to but not including the end.



Example: Code	**Explanation**	**Output**

`for number in range(5):`
`      print(number)`
	It has only an end value, 5. So by default, the start will be 0, the step size will be 1.	0, 1, 2, 3, 4.

`for number in range(2, 5):`
`      print(number)`
	It has start value 2 and end value, 5. So by default, the step size will be 1.	2, 3, 4.

`for number in range(0, 5, 2):`
`      print(number)`
	It has start value 0, end value, 5 and step size, 2	0, 2, 4.

Example: This program will print “Hi” for 5 times	
Code	**Output**

`for number in range(5):`
`     #codes inside for loop`
`     print("Hi")`
`     #codes inside for loop`
`#codes outside for loop`
	Hi
	Hi
	Hi
	Hi
	Hi

Explanation: Here, we are iterating using the range function with an end value of 5. So the range function returns a sequence [0, 1, 2, 3, 4] and “number” is the loop iterator which is going through all items of sequence serially from left to right.
During each iteration, the first checking is done to see whether any item is remaining and it is the loop terminating condition. If the checking gives True, then the item of the sequence is referred to the loop iterator, “number” and the loop body is executed. Inside the loop body, “Hi” is being printed. If the checking is False, then the for loop terminates and codes outside the loop start executing sequentially.

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/

Python Data structures and Files

Python has a few Python Specific Data Structures (List, Tuple, Dictionary) and non- Python Specific Data Structures (String, sets). All these data structures are quite sophisticated and help us to store a collection of values rather than a single one. For this course, we will discuss String, List, Tuple, and Dictionaries. Even though file reading is not part of the data structures, the ability to store and fetch previously-stored information is very essential for any language. SO we have covered it at the end of this chapter.

Strings

String is a sequence of ordered characters (alphabets-lower case, upper case, numeric values, special symbols) represented with quotation marks (single('), double(''), triple(' ' ', " " ")) at the beginning and end. The character order is always from left to right. For example: using single quotes ('CSE110'), double quotes ("CSE110") or triple quotes ('''CSE110''' or """CSE110"""), "Hello CSE110 students".

Single line String and Multi-line String
Strings can be made of a single line as well as multiple lines. For representing single lines, single(') or double('') quotation marks are used. Whereas for representing multiple lines, triple quotation marks (""") or three single quotation marks(''') **must** be used. The multiple line strings are usually called "Doc-Strings". Details about this will be discussed in the function chapter.

Example:	

Single line	`print("Loving CSE110 Course")`
	
	`Output: Loving CSE110 Course`

	`print('Loving CSE110 Course')`
	
	`Output: Loving CSE110 Course`

	`print('She said, "I love coding"')`
	
	`Output: She said, "I love coding"`

Multiple lines	`print("""We Love Coding in Python.`
	`But after mastering Python,`
	`should we call ourselves, "Snake charmers"`
	`or "Python programmers"?`
	`""")`
	
	`Output:`
	`We Love Coding in Python.`
	`But after mastering Python,`
	`should we call ourselves, "Snake charmers"`
	`or "Python programmers"?`

String printing special cases

1) In the last single line example, we had to print a pair of quotation marks as a part of the output. Since Python has 3 options (single('), double("), triple(' ' ', " " ") quotation) for representing String, we can easily print a pair quotation, using the alternative one. For example:

Code	Output
`print("I love 'Chocolates'")`	`I love 'Chocolates'`
`print('Craving for a "Dominos pizzas"')`	`Craving for a "Dominos pizzas"`

2) Assume, you have to print **She asked, "Aren't you late for work?".** Here you have to print a pair of quotation marks as well as one single quotation mark, so the previous way of string printing will not work here. This problem can be solved by using triple quotes(' ' ', " " "), representing special characters with a preceding backslash which is called an **escape character**. Using an escape character, or a preceding backslash tells the interpreter to consider the character following the backslash as a printable character. For example, putting **\"** will print ", putting **\'** will print ', and putting \\ will print \\.

To print our example, we can use multiple variations of escape characters. We can either put the entire string (along with the double quotes) inside a single quote and then use the escape character to print the single quote mark in the middle. Or in the opposite manner, we can put the entire string in double-quotes, and print the double quote mark in the string using escape characters. For example:

Code	Output
`print('''She asked, "Aren\'t you late for work?"''')`	`She asked, "Aren't you late for work?"`
`print('She asked, "Aren\'t you late for work?"')`	`She asked, "Aren't you late for work?"`
`print("She asked, \"Aren't you late for work?\"")`	`She asked, "Aren't you late for work?"`

Note: Details about escape characters and escape sequences are provided in the link below.
Link: https://docs.python.org/2.0/ref/strings.htmlEmpty String

Empty String

Strings represented with only two single(`''`) or two double(`""`) quotes with no characters in between are called **Empty String.** For example:

Code	Output

`print("")`
`print('')`
	Shows nothing in the output, because empty strings had nothing in between their quotation marks.

String indexing

For accessing individual characters within a string, python uses square brackets to enclose the index value or the position of that particular character
Indexing can be done in two ways:

1. Positive indexing:
   It is used to access the characters from the left side of a string. Positive indexing always starts from 0 (left-most character of the string) and ends at the last character of the string.

2. Negative indexing:
   It is used to access the characters from the right side of a string. Negative indexing always starts from -1 (rightmost character of the string) and ends at the first character of the string.

Positive indexing	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
	L	**o**	**v**	**i**	**n**	**g**		**C**	**S**	**E**	**1**	**1**	**0**
Negative indexing	**-13**	**-12**	**-11**	**-10**	**-9**	**-8**	**-7**	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**

Here, in the example above indexing of "Loving CSE110" is shown. The positive indexing starts at 0 with 'L' and ends at '0' with 12. The negative indexing starts at -1 with '0' and ends at 'L' with -13. If anything out of the valid range is tried to be accessed, the program will yield an IndexError. Additionally, if any other data type except an integer is entered as an index, TypeError will rise.

The basic String indexing structure
`string_name[ Index_value]`

For example:

Code	Output

`text = "Loving CSE110"`
`print(text[4])`
	`n`

`print(text[-4])`	`E`
`print(text[13])`	`IndexError`
`print(text[-15])`	`IndexError`
`print(text[1.5])`	`TypeError`

String slicing

Slicing is used for getting a substring of a particular string. In more general terms, it is used for accessing a particular range of characters. Colon(:) is used as a slicing operator.

Basic structure of slicing
`string_name[ beginning : end : step_size ]`
● Here, beginning: the index where slicing starts (inclusive). If not provided, by default starts from index 0. ● end: the index where slicing stops(Not inclusive). If not provided, by default includes the rest of the string after "beginning". ● Step: increment of the index value. If not provided, by default the value is 1.

Example:

Code	Output

`text = "Loving CSE110"`
`print(text[2:5])`
	`vin`
	prints from the 3nd character(index 2) to the 6th (index 5) character of the string

`print(text[4:-3])`	`ng CSE` prints from the 5th character (index 4) to the 3rd last character (index 10) of the string

`print(text[2:12:3])`	vgS1 prints from the 2nd character (index 2) to the 13th character (index 12) of the string, with a step size of 3.
`print(text[2:])`	ving CSE110 prints from the 3rd character(index 2) to the very last character of the string
`print(text[:-3])`	Loving CSE prints from the first character(index 0) to the 3rd last character (index 10) of the string
`print(text[-9:-2])`	ng CSE1 prints from the 9th last character of the string (index 4) to the 2nd last character (index 11) of the string

String Operators

Few of the basic operators discussed in the first chapter work quite differently while working with strings. In the table below, you will get an idea about their adapted working procedure with strings.

Operator	Meaning & Description	Example
+	Concatenation: Values are added from both side of the '+' operator	print("abc"+"de") Output: abcde
*	Repetition: a new string is created with the specified number of copies of the input string	print("abc"*4) Output: abcabcabcabc

string_name[Index_value]	String Indexing: as discussed above.	text = "Hello"
		print(text[1])
		Output: e

string_name[beginning : end : step_size]	String slicing: as discussed above.	text = "Hello Students"
		print(text[3:9])
		Output: lo Stu

in	Membership: Returns True if the first value is in the second. Otherwise, returns False.	"ph" in "elephant" Output: True

not in	Membership: Returns True if the first value is not in the second. Otherwise, returns False.	"ph" not in "elephant" Output: False
%	Format: used for formatting strings. Details will be discussed later.	Examples will be discussed later.

String Immutability

Immutable means once it has been created its value cannot be changed. So, each time we have to modify the values, we need to make a copy of the original one and make changes to the duplicate one. For example, the String's built-in methods. But it has an advantage that its elements can be accessed very fast. For example:

Code	Output

text = "Python"
text[3] = "K"
	TypeError: 'str' object does not support item assignment
	
	Explanation: here, we were trying to change the value of the 3rd index, 'h' to "K". Since python does not allow changes to immutable objects, we got an error in the output.

Here, from the example, we can see that Strings do not allow any kind of alteration to their values as Strings are immutable.

String methods

Python has a good number of built-in methods or functions to utilize Strings. Because of string immutable property, these methods do not change the original String rather returns a new String copy every time. Among these methods, the most frequently used ones have been mentioned in the table below.

Method name	Description	Example
len(string)	Returns the length of a string	print(len('Hello')) Output: 5

lower()	Returns a copy string with all lower case letters	text = 'Hello World'
		temp = text.lower()
		print(text)
		print(temp)

		Output: Hello World hello world

upper()	Returns a copy string with all upper case letters	`text = 'Hello World'`
		`temp = text.upper()`
		`print(text)`
		`print(temp)`
		
		Output:
		Hello World
		HELLO WORLD

strip()	Returns a copy string with all whitespace remove before and after letters	`text = '  BracU     CSE110     '`
		`temp = text.strip()`
		`print(text)`
		`print(temp)`
		
		Output:
		`    BracU     CSE110`
		`BracU     CSE110`

lstrip()	Returns a copy string with all whitespace remove before and after letters	`text = '  BracU     CSE110     '`
		`temp = text.lstrip()`
		`print(text)`
		`print(temp)`
		
		Output:
		`    BracU     CSE110`
		`BracU     CSE110`

rstrip()		`text = '  BracU     CSE110     '`
		`temp = text.rstrip()`
		`print(text)`
		`print(temp)`
		
		Output:
		`    BracU     CSE110`
		`    BracU     CSE110`

count(substring)	Returns total occurrence of a substring	`text = 'Bangladesh'`
		`temp = text.count('a')`
		`print(text)`
		`print(temp)`

		Output: Bangladesh 2

startswith(s
ubstring)
	Returns True if the String starts with given substring	text = 'Hello'
		temp =
		text.startswith('He')
		print(text)
		print(temp)
		
		Output:
		Hello
		True

endswith(sub
string)
	Returns True if the String ends with given substring	text = 'Hello'
		temp =
		text.startswith('hi')
		print(text)
		print(temp)
		
		Output:
		Hello
		False

find(substri
ng)
	Returns the index of first occurrence of substring	text = 'Bangladesh'
		temp = text.find('a')
		print(text)
		print(temp)
		
		Output:
		Bangladesh
		1

replace(olds
tring,
newstring)
	Replace every instance of oldstring with newstring	text = 'Hello'
		temp = text.replace('l',
		'nt')
		print(text)
		print(temp)
		
		Output:
		Hello
		Hentnto

Note: Details about String methods will be found in the link below.
Link: https://docs.python.org/2/library/stdtypes.html#string-methods

ASCII

ASCII stands for American Standard Code for Information Interchange. In order to store and manipulate data, it needs to be in binary values. Here, in ASCII 128 English characters and different symbols are represented with a 7-bit binary number, with a decimal value ranging from 0 to 127. For example, 65 is the ASCII value of the letter ‘A’, and 97 is the ASCII value of the letter ‘a’.

Two built-in Python functions ord() and chr() are used to make the conversions between ASCII values and characters. The ord() converts characters to ASCII values and chr() converts ASCII values to characters.

Example:	**Output**
`chr(98)`	`'b'`
`chr(66)`	`'B'`
`ord('a')`	`97`
`ord('%')`	`37`

Note: The ASCII chart will be found in the link below.
Link: http://www.asciitable.com/

Lists

Data structures are containers that organize and group data types together in different ways. A list in python is one of the most common and basic data structures that are written in square brackets "[]" with elements stored inside separated by commas. A list is also ordered and mutable. For example:

Code	Output

`list = ["Hi!", "Welcome", "to", "CSE", "110"]`
 	
`print(list)`
	`['Hi!', 'Welcome', 'to', 'CSE', '110']`

Mutability

The word "mutability" in python means the ability for certain types of data to be changed without recreating it entirely. It makes the program run more efficiently and quickly. For example:

Code	Output

`beatles = ["John", "Paul", "Alonzo", "Ringo"]`
 	
`#Before Modification`
`print(beatles)`
	`['John', 'Paul', 'Alonzo', 'Ringo']`

`beatles[2] = "George"`
 	
`#After Modification`
`print(beatles)`
	`['John', 'Paul', 'George', 'Ringo']`

Necessity of Lists

Lists don't need to be always homogeneous. They maintain the order of sequences whose values are not fixed. We can easily access and modify values inside a list. For example, adding value or removing is possible in a list. On the other hand, tuples are immutable and cannot be changed (you will get an explanation in the next section named "tuple"). Again, in python lists and strings are quite similar. We can use the "for" loop to iterate over lists, "+" plus operator in order to concatenate and use in a keyword to check if the sequence contains a value.

To access the Items of Lists:

1. Print the items in a list, we use index values:

Code	Output

`example_list = ["Kitkat", "Oreo", "Hersheys"]`
`print(example_list[2])`
	`Hersheys`

2. In a list, negative indexing means beginning from the end. Example: -1 refers to the last item, -2 refers to the second-last item, etc..

Code	Output

`example_list = ["Kitkat",`
`"Oreo","Hersheys"]`
`print(example_list[-1])`
`print(example_list[-2])`
	`Hersheys`
	`Oreo`

3. While specifying a range in a list, the return value will give a new list with the specified items:

Code	Output

`example_list = ["Hersheys","Kitkat", "Oreo",`
`"MIMI", "Cadbury", "Monchuri-Milk Candy"]`
 	
`print(example_list[2:5])`
	`['Oreo',`
	`'MIMI',`
	`'Cadbury']`

4. While specifying a range of negative indexes in a list, we can start the search from the end of a list. Given example returns the items from index -5 (included) to index -1 (excluded).

Code	Output

`example_list = ["Hersheys","Kitkat", "Oreo",`
`"MIMI", "Cadbury", "Monchuri-Milk Candy"]`
 	
`print(example_list[-5:-1])`
	`['Kitkat',`
	`'Oreo', 'MIMI',`
	`'Cadbury']`

5. By leaving out the start value, the range will start from the first item of the list. Again, by leaving out the end value, the range will go on to the end of the list.

Code	Output

`#Example-1)leaving out the start value:`
 	
`example_list = ["Hersheys","Kitkat", "Oreo",`
`"MIMI", "Cadbury", "Monchuri-Milk Candy"]`
 	
`print(example_list[:3])`
	`['Hersheys',`
	`'Kitkat',`
	`'Oreo']`

`#Example-2)leaving out the end value:`
 	
`example_list = ["Hersheys","Kitkat", "Oreo",`
`"MIMI", "Cadbury", "Monchuri-Milk Candy"]`
 	
`print(example_list[4:])`
	`['Cadbury',`
	`'Monchuri-Milk`
	`Candy']`

6. Since lists are mutable, we can change the items inside it.

Code	Output

`example_list = ["Hersheys","Kitkat", "Oreo",`
`"MIMI", "Cadbury", "Monchuri-Milk Candy"]`
 	
`print(example_list[2:5])`
	`['Oreo',`
	`'MIMI',`
	`'Cadbury']`

Basic Operations of List:

Example:	**Output**

`#1) To make a copy of items of the list into a new list, using [:] operator:`
 	
`example_list = ["Hersheys","Kitkat",`
`"Oreo", "MIMI", "Cadbury", "Monchuri-Milk`
`Candy"]`
 	
`duplicate_list = example_list [:]`
`print(duplicate_list)`
	`['Hersheys',`
	`'Kitkat', 'Oreo',`
	`'MIMI', 'Cadbury',`
	`'Monchuri-Milk`
	`Candy']`

`#2) The copied list remains unchanged when the item of original list gets modified:`
 	
`original_list = ["Hersheys","Kitkat",`
`"Oreo", "MIMI", "Cadbury", "Monchuri-Milk`
`Candy"]`
 	
`duplicate_list = original_list [:]`
`original_list[-1] = "Pran Mango Bar"`
 	
`print(original_list)`
`print (duplicate_list)`
	`['Hersheys',`
	`'Kitkat', 'Oreo',`
	`'MIMI', 'Cadbury',`
	`'`**`Pran Mango Bar`**`']`
	
	`['Hersheys',`
	`'Kitkat', 'Oreo',`
	`'MIMI', 'Cadbury',`
	`'`**`Monchuri-Milk`**
	`Candy``']`

`#3) Looping through list using for loop:`
 	
`example_list = ["Hersheys","Kitkat",`
`"Oreo", "MIMI"]`
 	
`for i in example_list:`
`   print(i)`
	`Hersheys`
	`Kitkat`
	`Oreo`
	`MIMI`

`#4) To check if an item exists in the tuple:`	`Yes,MIMI is in the`
	`list`

`example_list = ["Hersheys","Kitkat", "Oreo", "MIMI"]`
 	
`if "MIMI" in example_list:`
`    print("Yes,MIMI is in the list")`	

`#5) To know the number of items in a list:`
 	
`example_list = ["Hersheys","Kitkat", "Oreo", "MIMI"]`
 	
`print(len(example_list))`
	`4`

`#6) List membership Test:`
 	
`example_list = ["Hersheys","Kitkat", "Oreo", "MIMI"]`
 	
`print('Hersheys' in example_list)`
 	
`print('Cadbury' in example_list)`
	`True`
	
	`False`

Methods Associated with Lists:

Methods Associated with Lists	
L.append(e)	adds the object e to the end of L.
L.count(e)	returns the number of times that e occurs in L.
L.insert(i,e)	inserts the object e into L at index i.
L.extend(L1)	adds the items in list L1 to the end of L.
L.remove(e)	deletes the first occurrence of e from L.
L.index(e)	returns the index of the first occurrence of e in L.
L.pop(i)	removes and returns the item at index i in L. If i is omitted, it defaults to -1, to remove and return the last element of L.
L.sort()	sorts the elements of L in ascending order.
L.reverse()	reverses the order of the elements in L.

Built-in Methods of Lists with Examples:

Method	Code	Output

append()	`subjects = ['CSE', 'EEE', 'Civil']`
	`subjects.append('Mechanical')`
	`print(subjects)`
		`['CSE', 'EEE', 'Civil', 'Mechanical']`

count()	`subjects = ['CSE', 'EEE', 'Civil', 'CSE']`
	`x = subjects.count('CSE')`
	`print(x)`
		`2`

insert()	`subjects = ['CSE', 'EEE', 'Civil']`
	`subjects.insert(1, 'Mechanical')`
	`print(subjects)`
		`['CSE', 'Mechanical', 'EEE', 'Civil']`

extend()	`subjects = ['CSE', 'EEE', 'Civil']`
	`cars = ['Ford', 'BMW', 'Volvo']`
	`subjects.extend(cars)`
	`print(subjects)`
		`['CSE', 'EEE', 'Civil', 'Ford', 'BMW', 'Volvo']`

remove()	`subjects = ['CSE', 'EEE', 'Civil']`
	`subjects.remove('Civil')`
	`print(subjects)`
		`['CSE', 'EEE']`

index()	`subjects = ['CSE', 'EEE', 'Civil']`
	`x = subjects.index('Civil')`
	`print(x)`
		`2`

pop()	`subjects = ['CSE', 'EEE', 'Civil']`
	`subjects.pop(1)`
	`print(subjects)`
		`['CSE', 'Civil']`

sort()	`cars = ['Ford', 'BMW', 'Volvo']`
	`cars.sort()`
	`print(cars)`
		`['BMW', 'Ford', 'Volvo']`

reverse()	`subjects = ['CSE', 'EEE', 'Civil']`
	`subjects.reverse()`
	`print(subjects)`
		`['Civil', 'EEE', 'CSE']`

Slicing

We can understand slicing by visualizing the index to be in between the elements as shown below. How the slicing works have been explained in detail in the String chapter. The slicing mechanism works identically in all the data structures.

String	F	A	N	T	A	S	T	I	C
Positive indexing	0	1	2	3	4	5	6	7	8
Negative indexing	-9	-8	-7	-6	-5	-4	-3	-2	-1

Slicing elements of a list in python

If we want to access a range, we need two indices that will slice that portion from the list. For example:

Example:	**Output**

`#Declare a List named ‘my_list’`
 	
`my_list = ['F','A','N','T','A','S','T','I','C']`	

`#1) elements 3rd to 5th`
 	
`print(my_list[2:5])`
	`['N', 'T', 'A']`

`#2) elements beginning to 4th`
 	
`print(my_list[:-5])`
	`['F', 'A', 'N', 'T']`

`#3) elements 6th to end`
 	
`print(my_list[5:])`
	`['S', 'T', 'I', 'C']`

`#4) elements beginning to end`
 	
`print(my_list[:])`
	`['F', 'A', 'N', 'T', 'A', 'S', 'T', 'I', 'C']`

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/

Tuple

A tuple is a sequence of elements of any type and is immutable. In python, they are initialized inside parenthesis “()” instead of square brackets “[]”. To create a tuple, we simply have to do the following:

Code	Output

`example_tuple = ("banana", "mango", "apple")`
`print(type(example_tuple))`
	`<class 'tuple'>`

Tuples can hold values of any type. They can be homogeneous as well as heterogeneous. But we need to remember that once we declare those values, we cannot change them. For example:

Code	Output

`mixed_type = ('C','S','E', 1, 1, 0)`
 	
`for char in mixed_type:`
`    print(char,":",type(i))`
	`C : <class 'str'>`
	`S : <class 'str'>`
	`E : <class 'str'>`
	`1 : <class 'int'>`
	`1 : <class 'int'>`
	`0 : <class 'int'>`

`#Trying to change ‘E’ to ‘O’`
`mixed_type[2] = 'O'`
	`TypeError: 'tuple' object does not support item assignment`

Here, we are getting the error message because we are not allowed to change values inside a tuple.

Immutability of Tuple:

The word “immutability” in python means an object with a fixed value/id. Here “id” is the identity of a **location** of an object in **memory**. For example:

Code	Output
#1) Declare a Tuple named 'nth_tuple' `nth_tuple = (21,21,34,47)`	

#2) Items with the same value have the same id.
 	
`if id(nth_tuple[0]) == id(nth_tuple[1]):`
`    print("True")`
`print(id(nth_tuple[0]))`
	`True`
	
	`9756832`
	
	`9756832`

`print(id(nth_tuple[1]))`	

#3) Items with different values have different ids.
 	
`if id(nth_tuple[0]) == id(nth_tuple[2]):`
`   print ("True")`
`else:`
`    print ("False")`
 	
`print(id(nth_tuple[0]))`
`print(id(nth_tuple[2]))`
	`False`
	
	`9756832`
	
	`9757248`

#4) append function is not applicable in a tuple `nth_tuple.append(5)`	`AttributeError: 'tuple' object has no attribute 'append'`

Tuples are efficient:

Tuples provide no access to data values and are considered as faster than the lists. For example:

Code	Output

`#Execution time for tuple:`
 	
`import timeit`
 	
`print(timeit.timeit('x=(1,2,3,4,5,6,7,8,9)', number=100000))`
	`0.001523201004602015`

`#Execution time for List:`
 	
`print(timeit.timeit('x=[1,2,3,4,5,6,7,8,9]', number=100000)))`
	`0.00987649301532656`

In the above example, we have used the timeit() method imported from python library timeit to calculate the difference between the execution time of tuple and list. It is seen that tuple takes less time to execute than a list.

To access the Items of Tuple:

1. Print the third item in the tuple:

Code	Output

`example_tuple = ("Banana", "Mango", "Apple")`
`print(example_tuple[2])`
	`Apple`

2. In tuple, negative indexing means beginning from the end. Example: -1 refers to the last item, -2 refers to the second-last item, etc.:

Code	Output

`example_tuple = ("Banana", "Mango", "Apple")`
 	
`print(example_tuple[-1])`
`print(example_tuple[-2])`
	`Apple`
	`Mango`

3. While specifying a range in a tuple, the return value will give a new tuple with the specified items:

Code	Output

`example_tuple = ("Banana", "Mango",`
`"Apple", "Orange","Grape","Jackfruit")`
 	
`print(example_tuple[2:5])`
	`('Apple','Orange',`
	`'Grape')`

4. While specifying a range of negative indexes in a tuple, we can start the search from the end of a tuple. Given example returns the items from index -5 (included) to index -1 (excluded):

Code	Output

`example_tuple = ("Banana", "Mango",`
`"Apple", "Orange","Grape","Jackfruit")`
 	
`print(example_tuple[-5:-1])`
	`('Mango', 'Apple',`
	`'Orange', 'Grape')`

Tuple Unpacking:

To store elements of a tuple in separate variables is called unpacking. This operation allows us to take multiple return values from a function and store them in separate variables.

Basic example:

Code	Output

`example_tup = (3, 2, 1)`
`a, b, c = example_tup`
 	
`print("a:", a, "b:", b, "c:", c)`
	`a: 3 b:`
	`2 c: 1`

`#swapping values`
`b, c, a = example_tup`
`print("a:", a, "b:", b, "c:", c)`
	`a: 1 b:`
	`3 c: 2`

Advanced example: [Try this after function chapter]

Code	Output

```
def convert_days(days):
     years = days // 365
     months = (days - years * 365) // 30
     remaining_days = days - years * 365 - months *30
     return years, months, remaining_days
result = convert_days(4320)
print(result)
```
	(11, 10, 5)

```
years, months,days = result
print(years, months, days)
```
	11 10 5

```
years,months,days = convert_days(4000)
print(years, months, days)
```
	10 11 20

In the above example, a function is written which takes the number of days as input and returns the number of years, month, and remaining days. That is, the function returns a tuple of three elements. After that, we've split the tuple into three separate variables (years, months, days) each of which has its own value.

Basic Operations of Tuple:

Example:	**Output**

#1) Looping through tuple using for loop:
```
example_tuple = ("Banana", "Mango", "Apple",
"Orange","Grape","Jackfruit")

for fruit in example_tuple:
   print(fruit)
```
	Banana
	Mango
	Apple
	Orange
	Grape

#2) To check if an item exists in the tuple:
```
example_tuple = ("Banana", "Mango", "Apple",
"Orange","Grape","Jackfruit")

if "Grape" in example_tuple:
    print("Yes,Grape is in the tuple")
```
	Yes,Grape is in the tuple

#4) To create a tuple with single item, we need to put comma:
 	
`example_tuple = ("Grape",)`
`print(type(example_tuple))`
 	
`#NOT a tuple`
 	
`example_tuple = ("Grape")`
`print(type(example_tuple))`
	`<class 'tuple'>`
	`<class 'str'>`

#5) Because of its immutability, we cannot remove items in a tuple but, we can delete the tuple completely:
 	
`Delete_the_tuple = ("Anna", "Lukas", "Julia")`
`del Delete_the_tuple`
`print(Delete_the_tuple)`
	`NameError: name 'Delete_the_tuple' is not defined`

#6) For joining two tuples we can use '+' operator:
 	
`tuple1 = ("Anna", "Lukas","Julia")`
`tuple2 = (1, 2, 3)`
 	
`tuple3 = tuple1 + tuple2`
`print(tuple3)`
	`('Anna', 'Lukas', 'Julia', 1, 2, 3)`

Tuple Methods:

Method name	Description	Example

len()	Finds the number of items in the tuple	`example_tuple = ("Banana", "Mango", "Apple", "Orange")`
		`print(len(example_tuple))`
		
		`Output:`
		`4`

tuple()	Returns a tuple using double round-brackets.	`tuple_constructor = tuple(("Anna", "Lukas", "Julia"))`
		`print(tuple_constructor)`
		
		`Output:`

		`('Anna', 'Lukas', 'Julia')`

count()	Returns the number of times the value appears in the tuple.	`tuple_count =`
		`(53,32,78,12,2,3,6,7,53,53)`
		`number= tuple_count.count(53)`
		`print(number)`
		
		`Output:`
		`3`

index()	Searches for the first occurrence of the value 12, and returns its position.	`tuple_index =`
		`(53,32,78,12,2,3,6,7,53,53)`
		`x = tuple_index.index(2)`
		`print(x)`
		
		`Output:`
		`4`

enumerate()	It takes a list as a parameter and returns a tuple for each element in the list.	`grocery = ['bread', 'milk', 'butter']`
		`newGrocery = enumerate(grocery)`
		
		`# converting to tuple`
		`print(tuple(newGrocery))`
		
		`Output:`
		`((0, 'bread'), (1, 'milk'), (2,`
		`'butter'))`
		`((10, 'bread'), (11, 'milk'), (12,`
		`'butter'))`

To change tuple values:

Although tuples are unchangeable or immutable, we can still change the items of a tuple by converting the tuple into a list, change the list, and then, convert the list back to the tuple.

Example:	**Output**

`given_tuple = ("Banana", "Mango", "Apple")`
`to_list = list(given_tuple)`
`to_list[1] = "Kiwi"`
`new_tuple = tuple(to_list)`
 	
`print(new_tuple)`
	`('Banana',`
	`'Kiwi',`
	`'Apple')`

Dictionary

Dictionary

A dictionary is an unordered collection that consists of *key-value* pairs. It is python's inbuilt mapping type. It is similar to lists, but we know after studying so far that lists are sequential collections. Lists index their entries based on the position in the list which is sequenced from left to right, whereas dictionaries index their items **using *keys* and are unordered**.

The necessity of Dictionaries

Suppose, you want to specify a different index for a specific value as sometimes it makes sense to have keys for different elements. You cannot do that using lists as we know that list is a sequential collection. Think of a phonebook, if you want to collect a number, you search by name and get the number. Similarly, if you are asked to mention the page number of chapter 8 in a book. You can turn over the pages one by one and eventually find the heading Chapter 8 and then tell the page number. However, it is faster to go to the index, look for chapter 8, and tell the page number. In the above examples, the 'name' in the phonebook and the 'chapter 8' work as keys. So, the dictionary is needed for this kind of **direct searching**.

Key-value pair

As mentioned above, the dictionary is python's mapping type, here the mapping is from a *key* which can of any type that is immutable (unchangeable), to a *value* that can be any Python Data Object.

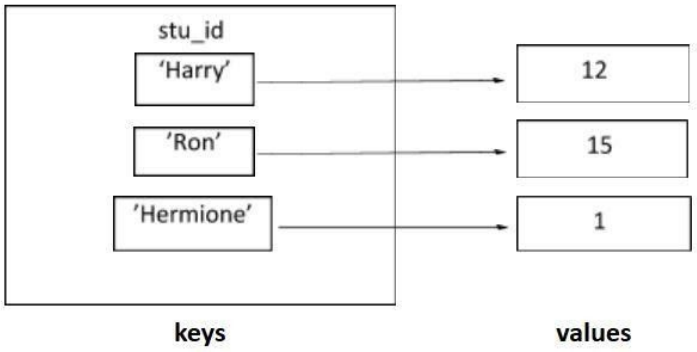
Creating a Dictionary
Example 1:

Code	Output

`stu_id = {}`
`stu_id ['Harry'] = 12`
`stu_id ['Ron'] = 15`
`stu_id ['Hermione'] = 1`
`print(stu_id)`
	`{'Harry': 12,`
	`'Ron': 15,`
	`'Hermione': 1}`

(Can also be set in a single line)
 	
`stu_id = {'Harry':12,'Ron':15,'Hermione':1}`
`print(stu_id)`
	`{'Harry': 12,`
	`'Ron': 15,`
	`'Hermione': 1}`

If we visualize the code, it looks like this.



Example 2:

Code	Output

`eng_to_fr = {}`
`eng_to_fr ['one'] = 'une'`
`eng_to_fr ['two'] = 'deux'`
`eng_to_fr ['three'] = 'trois'`
`print(eng_to_fr)`
	`{'one': 'une',`
	`'two': 'deux',`
	`'three': 'trois'}`

`eng_to_fr = { 'one': 'une', 'two': 'deux',`
`'three': 'trois'}`
`print(eng_to_fr)`
	`{'one': 'une',`
	`'two': 'deux',`
	`'three': 'trois'}`

In the above examples,

Key	Value	Key	Value
'Harry'	12	'one'	'une'
'Ron'	15	'two'	'deux'
'Hermione'	1	'three'	'trois'

So, dictionary literals have curly braces and have a list of **key: value** pairs which are separated by comma (,). Dictionary keys are **case sensitive**, same name but different cases of **the key** will be treated distinctly.

Empty Dictionary

Empty Dictionaries can be made using empty curly braces.

Code	Output

`abc = { }`
`print(abc)`
	`{ }`

Retrieving values

It does not depend on which order the pairs are written. To retrieve a value from the dictionary, it is the key that is needed. So you don’t have to think about which order they are.

Code	Output

`stu_id = {'Harry':12,'Ron':15,'Hermione':1}`
`value = stu_id['Hermione']`
`print(value)`
`print(stu_id['Harry'])`
	`1`
	`12`

You can get the same output by calling **get()** function**.**

Code	Output

`stu_id = {'Harry':12,'Ron':15,'Hermione':1}`
`value = stu_id.get('Hermione')`
`print(value)`
	`1`

The **get()** function returns the value of the key if the key exists in the dictionary. Otherwise, it shows **None** if the key is not present.

Code	Output

`stu_id = {'Harry':12,'Ron':15,'Hermione':1}`
`print(stu_id.get('Hermione'))`
`print(stu_id.get('Draco'))`
	`1`
	`None`

Dictionaries are Mutable

Dictionaries are mutable because their values can be modified. Dictionaries are something that needs constant changing - growing, shrinking, updating and deleting entries, etc. The examples below further illustrate how dictionary values are modified in place. For example:

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`print(box)`
`box['pen']= 2`
`print(box)`
	`{'pencil': 3, 'pen': 4,`
	`'eraser': 2}`
	
	`{'pencil': 3, 'pen': 2,`
	`'eraser': 2}`

`box = {'pencil':3,'pen':4,'eraser':2}`
`print(box)`
`box['pencil']= box['pencil']+3`
`print(box)`
	`{'pencil': 3, 'pen': 4,`
	`'eraser': 2}`
	`{'pencil': 6, 'pen': 4,`
	`'eraser': 2}`

Finding the length of Dictionary

We can find the total number of key-value pairs in the dictionary using the **len()** function.

Code	Output

`closet = {'shirt':3,'pant':4,'scarf':2}`
`num_items = len(closet)`
`print(num_items)`
	`3`

Looping through a Dictionary

We can loop through a dictionary by using for loop. For example, if you want to display all the key names from the dictionary:

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`for key in box:`
`     print(key)`
	`pencil`
	`pen`
	`eraser`

And, for printing all the values in the dictionary:

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`for key in box:`
`    print(box[key])`
	`3`
	`4`
	`2`

The values of the dictionary can also be returned by using **the values()** function.

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`for val in box.values():`
`    print(val)`
	`3`
	`4`
	`2`

We can also loop through both keys and values, by using **items()** function.

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`for key,val in box.items():`
`    print(key,val)`
	`eraser 2`
	`pencil 3`
	`pen 4`

Using the **key()** function, we get the keys of the dictionary as a list.

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`key_values = box.keys()`
`print(key_values)`
	`dict_keys(['pencil',`
	`'pen', 'eraser'])`

If you update the dictionary, then the updated list of keys will be displayed.

Code	Output

`box = {'pencil':3,'pen':4,'eraser':2}`
`key_values = box.keys()`
`box['cutter'] = 1`
`print(key_values)`
	`dict_keys(['pencil',`
	`'cutter', 'pen',`
	`'eraser'])`

Determining if a key exists in Dictionary

For checking whether a key is in the dictionary or not, we use the membership operator, **in**.

Code	Output

`closet = {'shirt':3,'pant':4,'scarf':2}`
`if 'scarf' in closet:`
`     print('Yes')`
	`Yes`

`closet = {'shirt':3,'pant':4,'scarf':2}`
`print('scarf' in closet)`
	`True`

`closet = {'shirt':3,'pant':4,'scarf':2}`
`print('shoes' in closet)`
	`False`

Adding elements

If we want to add a new element to the dictionary, we can do it by using a new key and providing value to it.

Code	Output

`closet = {'shirt':3,'pant':4,'scarf':2}`
`closet['shoes'] = 1`
`print(closet)`
	`{'shirt': 3, 'pant':`
	`4, 'scarf': 2,`
	`'shoes': 1}`

You can also insert an element to your dictionary using update() method.

Code	Output

`closet={'shirt':3, 'pant':4,`
`'scarf':2}`
`closet.update({'shoes':1})`
`print(closet)`
	`{'shirt': 3, 'pant': 4,`
	`'scarf': 2, 'shoes': 1}`

Removing elements

To remove an element from the dictionary, we use **del** keyword with the specified key name.

Code	Output

`closet = {'shirt':3,'pant':4,'scarf':2}`
`del closet['scarf']`
`print(closet)`
	`{'shirt': 3, 'pant': 4}`

The **del** keyword can also be used to remove the whole dictionary.

Code	Output

`closet = {'shirt':3,'pant':4,'scarf':2}`
`del closet`
`print(closet)`
	`NameError: name 'closet' is not defined`
	
	Here, trying to print the dictionary closet is showing error as the dictionary has been deleted.

Sorting Dictionary

You can also sort the elements of the dictionary by values or keys by using **sorted().**

By keys,

Code	Output

`box = {'cutter':3,'pen':4,'eraser':2}`
`ksort = sorted(box.keys())`
`print(ksort)`
	`['cutter', 'eraser', 'pen']`

By values,

Code	Output

`box = {'cutter':3,'pen':4,'eraser':2}`
`ksort = sorted(box.values())`
`print(ksort)`
	`[2, 3, 4]`

Copying Dictionary

We can also make a copy of the dictionary by using the copy**()** function. For example: suppose, we have a dictionary of a library. The library includes both fiction and non-fiction genres. But we want to see fiction and non-fiction parts separately.

Code	Output

`library = {'sci-fi': 25,'mystery':17,'mythology':12,'biography':22, 'history':20}`
`fictions = {'sci-fi': 25,'mystery':17,'mythology':12}`
`nonfictions = {'biography':22,'history':20}`
 	
`fiction_dict = fictions.copy()`
`non_fiction_dict  = nonfictions.copy()`
`print(fiction_dict)`
`print(non_fiction_dict)`
	`{'sci-fi': 25, 'mystery': 17, 'mythology': 12}`
	`{'biography': 22, 'history': 20}`

Now look at the above example, we have declared the items of fictions and non-fictions genres in separate dictionaries named `fictions` and `nonfictions`. Then the items of both the fictions and non-fictions dictionaries are copied into the new dictionaries named `fiction_dict` and `non_fiction_dict` using the **copy()** function.

So now if you want to check both genres separately, you need not look at the main dictionary library. You just print `fiction_dict` and `non_fiction_dict` and see how many books are available in each category.

String, List, Tuple, Dictionary, Range

When to use which data type?

Before knowing when to use which data types, we need to know the difference between them. The table below illustrates their difference.

Data type	Type of elements	Mutability	Indexed By	Supports “in” operator	Supports “ +” and “*” operator
String	Characters	Immutable	integers	Yes	Yes
List	Any type	Mutable	Integers	Yes	Yes
Tuple	Any type	Immutable	Integers	Yes	Yes
Dictionary	Values any type, but keys immutable type	Mutable	Any immutable data type	Yes	No
Range Function	Integers	Immutable	Integers	Yes	Yes

Advantage-Disadvantage of List:

Due to lists mutability, it is more popular among the programmer’s community than Tuple. For example, we can use it to store any values of any data types while still building it. But due to the same characteristic (mutability), it cannot be used as a key for dictionaries and it is prone to aliasing problems. So, when we do not want the original list to be modified, copies of the original list are made before making any kind of modifications. For example, the slicing method makes a new copy of the original list automatically every time before returning the modified list as a return value.

Advantage-Disadvantage of Tuple:

Since tuples are immutable, they do not have any side effects as well as any aliasing problems. For example, for lessening the chances of any possible aliasing problems, tuples can be used to pass values as an argument to functions. Additionally, tuples can hold any data types as its elements. But for a tuple to work as a dictionary key, its elements have to be immutable.

Advantage-Disadvantage of String:

Unlike tuples and lists, strings can only have characters as its elements and these characteristics make strings less flexible. But, in Python3, strings have many built-in methods compared to other data types which make its use quite effortless. Additionally, due to the string's immutability, it has no side effects and it can be used as dictionary keys.

Advantage-Disadvantage of Dictionaries:

Dictionary keys must be unique and immutable data types. However, its values can be any data type including the user-defined custom object types (will be discussed later in this course) and there are no mandatory restrictions for its values to be unique. Since dictionaries are mutable, they can have aliasing problems. For solving this, duplicates of the original dictionary are made, before making any kind of modifications.

Advantage-Disadvantage of Range:

Range function returns an immutable sequence of integers (actually returns a range object) which can be converted into a list. It can also be used in "for loop iteration" (iteration a fixed number of times). For iterating, before the iteration, the "immutable sequence" is automatically converted into a "list".

Code	Output

#1)converting immutable sequence of integers to list
 	
`odd_numbers = list(range(1,12,2))`
`print(odd_numbers)`
	`[1, 3, 5, 7, 9, 11]`

#2)used in for loop iteration (manually converting it into list)
 	
`for num in list(range(1,12,2)):`
`          print(num)`
	1
	3
	5
	7
	9
	11

#3)used in for loop iteration (automatically converting it into a list)
 	
`for num in range(1,12,2):`
`          print(num)`
	1
	3
	5
	7
	9
	11

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/

Functions

Function

The function is a block of effectively grouped related codes which is utilized to execute a single, connected action and can be reused multiple times. For example, print() is a Python built-in function, which we use multiple times by calling it with its name. Similar to this one, we can make custom functions and these are called “user-defined functions”.

Defining a function

The basic structure of a Function

```
def function_name (parameter_list):
        """docstring"""
        Body_of_function

        return [expression]
```

Example of function

```
def greetings():
     #function body starts
     """This function prints greetings"""
     print("Hello Students")
     print("I hope you guys are loving Python")
     #function body ends
```

```
def make_cube(number):
     #function body starts
     '''This function makes cube of the
     given number in the parameter'''
     print(number**3)
     #function body ends
```

1. **Def**: It is a Python reserved keyword that is used to inform Python that a function is being defined or started.

2. **Name_of_function:** It is used for referring or calling to the function. For example, in the above code, `make_cube` is the function name. Function names can be anything except the Python

reserved keywords. Similar to variable naming, it has to maintain certain rules.

Function names-

- can have letters (a-z), digits(0-9), and underscores.
- Words in a function name should be separated by an underscore
- cannot begin with digits.
- cannot have whitespace and special signs (e.g.: +, -, !, @, $, #, %.)
- are case sensitive meaning ABC, Abc, abc are three different variables.
- cannot be a reserved keyword for Python.

Note: For details read from the link below.
Link: https://www.python.org/dev/peps/pep-0008/#function-and-variable-names

3. **Parameters (arguments):** We can have an empty parameter (no variables within the parentheses) or multiple parameters separated with commas. Parameter_list is used to pass values(or inputs) in the function. These are optional. Here, in the above example, the number is a parameter.

4. **Docstring:** This triple quoted string is written in the first statement of the function and contains documentation or the purpose of the function. It is **optional**. But maintaining docstring is a convention. Python has built-in functions to access these docstrings.

5. **Body_of_function:** Within the function, codes related to producing a certain result are written in an indented manner in the function body. It can have multiple sequential or conditional statements.

Function call/invocation

If we run the previous two examples, we will get no output or result. In order to invoke the function, only defining it is not enough. We need to call the function with its function name, accompanied by the parentheses. While invoking/calling, the parentheses can have no parameters or multiple parameters separated by commas depending on the structure of the defined function. Functions can be called directly or from other functions or straight from the Python prompt.

Code	Output

```
def greetings():
     """This function prints greetings"""
     print("Hello Students")
     print("I hope you are loving Python")

#driver code
#calling the function with an empty parameter
greetings()
```
	```
	Hello Students
	I hope you are
	loving Python
	```

```
def make_cube(number):
     '''This function makes cube of the
     given number in the parameter'''
     print(number**3)

#driver code
#calling the function with input 2
makeC_cube(2)
#calling the function with input 3
make_cube(3)
```
	```
	8
	27
	```

Parameters (arguments):
Parameters: Parameters are the variables declared in the function.
Arguments: Arguments are the values passed through the parameters when calling the function.

1. **Positional arguments:** During the function call, all the arguments must be provided and they have to be in the same order as the function definition. Here **position or order** is important.

Code: Function definition
```
def print_info(name, age, height):
     print("Name:", name, "Age:", age,"Height:", height, "cm")
```

Code: Function call	Output

```
#calling with proper order

print_info("John", 20, 167)
```
	`Name: John Age: 20 Height: 167`

Here, “John” is being mapped to name, 20 is being mapped to age, and 167 is being mapped to height.	

Code: Function call	Output

`#calling with the wrong order`
`print_info( 20, 167, "John")`
	`Name: 20 Age: 167 Height: John`

Here, 20 is being mapped to name, 167 is being mapped to age, and “John” is being mapped to height. Since the order of position during the function call was wrong, so our mapping was also wrong. So, finally, the output of the code was wrong.	

2. **Default arguments:**

When a function has one or multiple arguments with default values in the function definition, then a few of arguments (with default values) can be **excluded** during the function call. Then, the excluded arguments by default take the values provided in the function definition. If no default values are set in the function for the excluded arguments, then Python raises an error, TypeError.

Code: Function with default argument

```
def greetings(quizzes, name = "students" ):
     """This function prints greetings"""
     print("Hello " + name + "!")
     print("Hope you are loving Python.")
     print("You will have " + str(quizzes) + " this term")
     print("===========================")
```

Code: Function call	Output

`#1)calling the function with values john and 10`
 	
`greetings(10, "John")`
	`Hello John!`
	`Hope you are loving Python.`
	`You will have 10 this term`
	`===========================`

Here, 10 is being mapped to the quizzes variable, and “John” is being mapped to the name variable, overwriting the default value,”students”.	

Code: Function call	Output
#2)calling the function with only 10, skipping the name greetings(10)	Hello students! Hope you are loving Python. You will have 10 this term ==========================
Here, 10 is being mapped to the quizzes variable and since no value has been provided for the name variable, the default value ”students” is used for the name variable.	

Code: Function call	Output
#3)calling the function with an empty parameter greetings()	TypeError: greetings() missing 1 required positional argument: 'quizzes'
Here, the function had been tried to call with an empty parameter. But, since quizzes do not have any default values defined in the function, the Python raises an error.	

Another thing to keep in mind, the non-default argument(parameter with no defined values) must come before the default argument (parameter with defined values). Otherwise, Python will raise an error, SyntaxError.

Code	Output

def greetings(name = "students", quizzes):
     """This function prints greetings"""
     print("Hello " + name + "!")
     print("Hope you are loving Python.")
     print("You will have " + str(quizzes) + " this term")
     print("==========================")
	SyntaxError: non-default argument follows default argument

3. **Keyword arguments:** During the function call, all the arguments must be provided, but they can be in **random order.** Here position or order is not important.

Code: Function definition

`def greetings(quizzes, name = "students" ):`
`    """This function prints greetings"""`
`    print("Hello " + name + "!")`
`    print("Hope you are loving Python.")`
`    print("You will have " + str(quizzes) + " this term")`
`    print("==========================")`

Code	Output

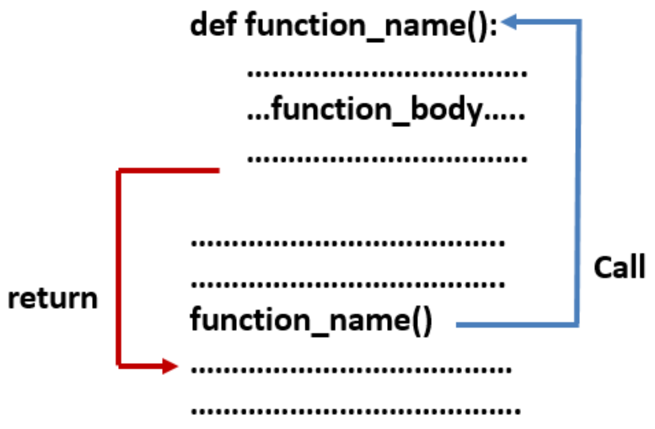
`#1)calling with keyword argument`
 	
`greetings(name = "John", quizzes = 10)`
	`Hello John!`
	`Hope you are loving Python.`
	`You will have 10 this term`
	`==========================`

`#2)calling without keyword argument`
 	
`greetings("John", 10)`
	`TypeError: can only concatenate str (not "int") to str`
	
	Here, "John" is being mapped to the quizzes variable, and 10 is being mapped to the name variable, overwriting the default value,"students".

The *return* Statement

When a function is called, before exiting the function body, it returns a value to the point where the function was called from. **If no values are returned from the function body, then by default it returns None (no values at all)**. In order to see the default value, we need to print the function.

By the return statement, results from the function can be passed to the main program. These results provided by the function can be saved for later use in the program, or it can be used to make a more complex expression. While being called, a function can only execute the return statement once. Printing inside a function is mainly done for human consumption (people to see the output).



Code	Output

```
def check_even(num):
     if num%2 == 0:
         print("even")
     else:
         print("Odd")


print(check_even(3))
```
	Odd
	None

```
def check_even(num):
     if num%2 == 0:
         print("even")
     else:
         print("Odd")

     return

answer = check_even(3)
print(answer)
```
	Odd
	None

```
def check_even(num):
     if num%2 == 0:
         return "even"
     else:
         return "Odd"

answer = check_even(3)
print(answer)
```
	Odd

def check_even(num):
    if num%2 == 0:
        return "even"
        return "2nd return in if"
    else:
        return "Odd"
        return "return in function body"
 	
print(check_even(2))
	even

def check_even(num):
    return "return in function body"
    if num%2 == 0:
        return "even"
    else:
        return "Odd"
 	
print(check_even(2))
	return in function body
	
	Here, the first return statement is being executed.

Necessity of Functions

1) Code reusability: Suppose we want to calculate the area of 3 different lands. What you have learned so far, for calculating the area, we need to write the same code 3 times. But using a function we can do it only once and reuse it.

Code	Output

def calculateArea(length, width):
    '''This function calculates area of a land
    with the given parameters length and
    width'''
 	
    print(length*width)
 	
calculateArea(100, 50)
calculateArea(150, 40)
calculateArea(200, 30)
	5000
	6000
	6000

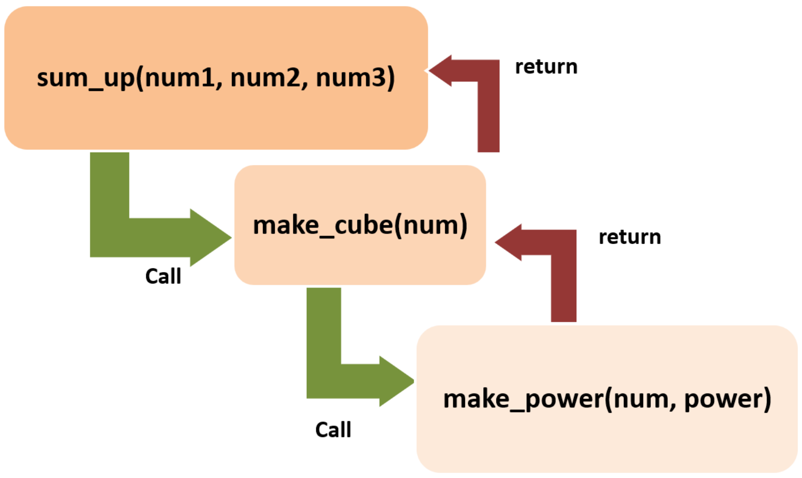
2) Minimize writing identical code within a program.
3) Better modularity: Large programs might have hundreds or thousands of lines of code. The function helps us divide those codes into small easily manageable chunks of codes. So, our code becomes very easy to debug (correcting problems within a code).
4) Easy to manage: The more modular our code is, the more manageable and organized it is.
5) Easily used inside a program: Functions made it possible to reuse codes easily inside complex computations without making it more lengthy.

6) Provides decomposition(breaking a huge program into small related reusable code chunks) and abstraction(hides unnecessary information, for example, a function is calculating roots. every time we use it, we do not need to know or see 100 lines of codes)

Anonymous Functions
Anonymous functions are written with the lambda keyword. If you are interested to learn more about Lamba read the details from the provided link.
Link: https://docs.python.org/3/reference/expressions.html#lambda

Functional decomposition:

When a large problem is broken down into smaller sub-problems, it is called Functional decomposition.
In Python, all the functions can be called from other functions. The function calling another function is known as "Calling function", whereas the function which is being called, is known as "Called function"
Here, in the example, our problem has been solved using multiple functions and inner function calls.



Code	Output

```
def make_power(num, power):
     result = num ** power
     return result

def make_cube(num):
     result = make_power(num, 3)#calling the make_power
     return result

print(make_cube(5))
```
	125

```
def make_power(num, power):
     result = num ** power
     return result

def make_cube(num):
     result = make_power(num, 3)
     return result

def sum_up(num1, num2, num3):
     s1 = make_cube(num1)
     s2 = make_cube(num2)
     s3 = make_cube(num3)
     return s1 + s2 + s3

print(sum_up(1, 2, 3))
```

```
36
```

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/

Searching

Searching is occasionally dependent on Sorting. We sort and store data in a particular order for finding it quickly during a search procedure. We will learn two basic searching techniques: Sequential search(searches through a sequence) and Binary search(search through a sorted random-access sequence) in this chapter.

Sequential/Linear Search

The easiest way to search for an item in a sequence is to start at the beginning and keep going through each item at a time for finding a match with the key. If the match is found, we stop and return the location/index of the item in the sequence. If we go through all the items of the list and no match is found, that means the key was not in the sequence. This process is called "Sequential/Linear Search".
For n element sequence, the worst-case requires n comparisons. The benefit of this searching technique is it can work with unsorted data and it does not require random access.

The following shows how to sequentially search for a key in a list.

Linear/Sequential search	Output

`def sequential_search(data, key):`
`    for index in range(len(data)):`
`        if key == data[index]:`
`            return index #index if found`
`    return -1 #sentinel if not found`	

#Key exists in the data
`data = [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300]`
`sequential_search(data, 100)`
	6

#Key does not exist in the data `sequential_search(data, 30)`	-1

Binary Search

If the data in the sequence is already sorted, and the sequence supports random access (it can be accessed with an index number), then we can significantly improve the search performance by using binary search.

We're already quite familiar with this search technique — we use something similar when we look up a word in a dictionary. We see if the item is in the middle of the list. If so, we've found it. If not, we check if the item is larger or smaller than the item we are looking for. If it's larger, then we look at the right half of the list (to the right of the middle element). Or else we look at the left half (to the left of the middle element). Then we again look at the middle element of the half we have chosen and repeat the process. We continue dividing up the list and repeat the process, until either we find the item, or we run out of data to search (ie., we cannot divide the data anymore, in which case we can say the item was not in the list).

The following shows how to search for a key in a list with sorted elements.
Input: [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300].
Key: 15 **(Key exists in the data)**

```
Current section: [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300]
================================
Left: 0 , Right: 9
Mid index: 4 , Mid value: 20
15 < 20 ||| Moving left
Current section: [-5, 5, 10, 15]
================================
Left: 0 , Right: 3
Mid index: 1 , Mid value: 5
15 > 5 ||| Moving right
Current section: [10, 15]
================================
Left: 2 , Right: 3
Mid index: 2 , Mid value: 10
15 > 10 ||| Moving right
Current section: [15]
================================
Left: 3 , Right: 3
Mid index: 3 , Mid value: 15
15==key, return Mid index (FOUND THE KEY)
```

The following shows how to search for a key in a list with sorted elements.
Input: [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300].
Key: 16 **(Key does not exist in the data)**

```
Current section: [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300]
================================
Left: 0 , Right: 9
Mid index: 4 , Mid value: 20
16 < 20 ||| Moving left
Current section: [-5, 5, 10, 15]
================================
Left: 0 , Right: 3
Mid index: 1 , Mid value: 5
16 > 5 ||| Moving right
Current section: [10, 15]
================================
Left: 2 , Right: 3
Mid index: 2 , Mid value: 10
16 > 10 ||| Moving right
Current section: [15]
================================
Left: 3 , Right: 3 (STOP)
Mid index: 3 , Mid value: 15
16 > 15 ||| Moving right
left>right  return -1 (KEY NOT FOUND)
================================
```

Binary search

```
def binary_search(data, size, key):

     left = 0
     right = size - 1
     while left <= right:
         mid_index = (left+ right)//2
         mid_val = data[mid_index]
         if key == mid_val:
             return mid_index
         elif key > mid_val:
             left = mid_index + 1
         else:
             right = mid_index - 1

      return -1 #sentinel if not found

data = [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300]
binary_search(data, len(data), 5)
```

Sorting

Arranging data in an ordered sequence is called "**sorting**". This order can be done in two ways: ascending and descending.

Ascending order: arranged from the smallest to the largest number.
Descending order: arranged from the largest to the smallest number.

Python has 2 built-in functions called sort() and sorted(). Besides these, we will look into two sorting algorithms named Selection sort and Bubble sort.

The sorted() function

The sorted() function makes a copy of the original list and returns the sorted list. It does not modify the original list.

Syntax of sorted() Function
`sorted(iterable, key, reverse=False)`
Parameters
Iterable: Sequence data type (string, list, tuple) or collection data type(dictionary, set) **Key(optional):** It is a function that works like a key and helps with custom sorting. The function named passed here can be built-in or custom-made. **Reverse(optional):** Its default value is False. For False, it sorts in the ascending order, and for True, it sorts in descending order. **Return Type:** Returns a sorted copy of the original list.

Example:	**Output**

#1) Ascending sort of a list without the optional reverse
 	
`numbers = [10, 20, 70, 60, 40]`
`sorted_numbers = sorted(numbers)`
`print("Sorted:", sorted_numbers)`
`print("Original:", numbers )`
	`Sorted: [10, 20, 40, 60, 70]`
	
	`Original: [10, 20, 70, 60, 40]`

#2) Ascending sort of a tuple without the optional reverse	`Sorted: [10, 20, 40, 60, 70]`

```
numbers = (10, 20, 70, 60, 40)
sorted_numbers = sorted(numbers)
print("Sorted:", sorted_numbers)
print("Original:", numbers )
```

```
Original: (10,
20, 70, 60, 40)
```

#3) Ascending sort of a String without the optional reverse. For the string, sorting is done depending on the ASCII values.

```
unsorted_chars = "CSE110 Rocks"
sorted_chars = sorted(unsorted_chars.lower(),
reverse = False)
print("Sorted:", sorted_chars)
print("Original:", unsorted_chars)
```

```
Sorted: [' ',
'0', '1', '1',
'c', 'c', 'e',
'k', 'o', 'r',
's', 's']

Original: CSE110
Rocks
```

#4) Descending sort of a String with the optional reverse. For the string, sorting is done depending on the ASCII values.

```
unsorted_chars = "CSE110 Rocks"
sorted_chars = sorted(unsorted_chars.lower(),
reverse = True)
print("Sorted:", sorted_chars)
print("Original:", unsorted_chars)
```

```
[Sorted: ['s',
's', 'r', 'o',
'k', 'e', 'c',
'c', '1', '1',
'0', ' ']

Original: CSE110
Rocks
```

#6) Descending sort of a Dictionary with the optional reverse. For the Dictionary, sorting is done depending on key values.

```
dict_1 = {1:"A", 100:"B", -5:"C", 44:"D", 2:"E"}
sorted_keys = sorted(dict_1, reverse = True)
print("sorted_keys:",sorted_keys)

print("Printing values according to the sorted
keys:", end=" ")
for key in sorted_keys:
     print(dict_1[key], end = " ")
```

```
sorted_keys:
[100, 44, 2, 1,
-5]

Printing values
according to the
sorted keys: B D
E A C
```

#7) Ascending sort of a list without the optional reverse. Here, the key is the built-in len() function. So, the sorting is done depending on the length of the string.

```
words = ['abcd', 'ef', 'g', 'hjklmno', 'pqr']
sorted_words = sorted(words, key = len)
print("Sorted:", sorted_words)
print("Original:", words)
```

```
Sorted: ['g',
'ef', 'pqr',
'abcd',
'hjklmno']

Original:
['abcd', 'ef',
'g', 'hjklmno',
'pqr']
```

	s

#7) Ascending sort of a list with the optional reverse. Here, the key is the built-in len() function. So, the sorting is done depending on the length of the string.
 	
`words = ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
`sorted_words = sorted(words, key = len, reverse = True)`
`print("Sorted:", sorted_words)`
`print("Original:", words)`
	`Sorted: ['hjklmno', 'abcd', 'pqr', 'ef', 'g']`
	`Original: ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`

The sort() function

The sort() function modifies the original list. Unlike sorted(), <u>it can only be used with lists.</u>

Syntax of sort() Function
`List_name.sort(key,  reverse = False)`
Parameters
Key(optional): It is a function that works like a key and helps with custom sorting. The function named passed here can be built-in or custom-made. **Reverse(optional):** Its default value is False. For False, it sorts in the ascending order, and for True, it sorts in descending order. **Return Type:** None

Example:	**Output**

#1) Ascending sort of a list without the optional reverse
 	
`numbers = [10, 20, 70, 60, 40]`
`print("Before:", numbers)`
`numbers.sort()`
`print("After:", numbers)`
	`Before: [10, 20, 70, 60, 40]`
	
	`After: [10, 20, 40, 60, 70]`

#2) Descending sort of a list with the optional reverse.
 	
`numbers = [10, 20, 70, 60, 40]`
	`Before: [10, 20, 70, 60, 40]`
	
	`After: [70, 60,`

`print("Before:", numbers)`
`numbers.sort(reverse = True)`
`print("After:", numbers)`
	`40, 20, 10]`

#3) Ascending sort of a list without the optional reverse. Here, the key is the built-in len function. So the sorting is done depending on the length of the string.
 	
`words = ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
`print("Before:", words)`
`words.sort(key = len)`
`print("After:", words)`
	`Before: ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
	
	`After: ['g', 'ef', 'pqr', 'abcd', 'hjklmno']`

#4) Descending sort of a list with the optional reverse. Here, the key is the built-in len function. So the sorting is done depending on the length of the string.
 	
`words = ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
`print("Before:", words)`
`words.sort(key = len, reverse = True)`
`print("After:", words)`
	`Before: ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
	
	`After: ['hjklmno', 'abcd', 'pqr', 'ef', 'g']`

Read the details about the built-in sorting from the provided link.
Link: https://docs.python.org/3/howto/sorting.html

Selection sort

For selection sort, in the ascending order, we need to find the minimum value of the given sequence and swap it with the first element of the sequence. Then, we need to find the second minimum from the rest of the data (starting from the 2nd index) and swap it with the 2nd element of the sequence. Then, we need to find the 3rd minimum from the rest of the data (starting from the 3rd index) and swap it with the 3rd element of the sequence. We need to continue this process until all the elements of the sequence have moved to their proper position.
At each iteration, the sequence is partitioned into two sections. The left section is sorted and the right section is being processed. Each iteration, we move the partition one step to the right, until the entire sequence has been processed.

The following shows the sequence of steps in sorting the sequence [17 3 9 21 2 7 5].
In the example below, the "|" symbol shows where the partition is at each step.
Input: 17 3 9 21 2 7 5

Step 1: | 17 3 9 21 2 7 5 << minimum is 2, exchange with 17 (index0’s value)
Step 2: 2 | 3 9 21 17 7 5 << minimum is 3, no exchange needed
Step 3: 2 3 | 9 21 17 7 5 << minimum is 5, exchange with 9 (index2’s value)
Step 4: 2 3 5 | 21 17 7 9 << minimum is 7, exchange with 21 (index3’s value)
Step 5: 2 3 5 7 | 17 21 9 << minimum is 9, exchange with 17 (index4’s value)
Step 6: 2 3 5 7 9 | 21 17 << minimum is 17, exchange with 21 (index5’s value)
Step 7 : 2 3 5 7 9 17 | 21 << STOP

For each iteration, we only consider the right section of the partition for finding the minimum value as the left side is already sorted. Lastly, in the last step, only one element remains which is the minimum. So sorting is unnecessary. So, for selection sort, size-1 times iteration is needed.

For descending order just find the maximum value instead of the minimum value.

Selection Sort

```
numbers = [17, 3, 9, 21, 2]
print("Before Sorting:", numbers)#size/length is = 5

for index1 in range(0, len(numbers)-1): #0,1,2,3 (Iteration 4)
     min_val = numbers[index1]
     min_index = index1
     #finding the minimum value from partition to rest of data
     #partition/index is moving to right

     for index2 in range(index1+1, len(numbers)):
          if numbers[index2] < min_val:
                min_val = numbers[index2]
                min_index = index2

     #swapping partition's right value with the min value
     temp =  min_val
     numbers[min_index] =  numbers[index1]
     numbers[index1] = temp

print("After Sorting:", numbers)
```

For simulation(visualization) of the algorithms use this link below.
Link: https://visualgo.net/en/sorting

Bubble sort

For Bubble sort, in each iteration, adjacent elements are compared. If the adjacent elements are in the wrong order, then they are swapped. It has been named “Bubble sort” because of the way smaller or larger elements "bubble" to the top(Left side) of the list. As the number of iterations increase, the number of comparisons decreases.

The following shows the sequence of steps in sorting the sequence [17 3 9 21 2 7 5].
Input: 17 3 9 21 2 7 5

```
Before Sorting: [17, 3, 9, 21, 2]
Iteration: 0
Total compares: 4
[17, 3, 9, 21, 2] compare: 17 > 3 Swap places
[3, 17, 9, 21, 2] compare: 17 > 9 Swap places
[3, 9, 17, 21, 2] compare: 17 < 21 No swap
[3, 9, 17, 21, 2] compare: 21 > 2 Swap places
[3, 9, 17, 2, 21]
============================
Iteration: 1
Total compares: 3
[3, 9, 17, 2, 21] compare: 3 < 9 No swap
[3, 9, 17, 2, 21] compare: 9 < 17 No swap
[3, 9, 17, 2, 21] compare: 17 > 2 Swap places
[3, 9, 2, 17, 21]
============================
Iteration: 2
Total compares: 2
[3, 9, 2, 17, 21] compare: 3 < 9 No swap
[3, 9, 2, 17, 21] compare: 9 > 2 Swap places
[3, 2, 9, 17, 21]
============================
Iteration: 3
Total compares: 1
[3, 2, 9, 17, 21] compare: 3 > 2 Swap places
[2, 3, 9, 17, 21]
============================
After Sorting: [2, 3, 9, 17, 21]
```

Bubble Sort

```
numbers = [17, 3, 9, 21, 2] #size/length is = 5
print("Before Sorting:", numbers)

for idx1 in range(0, len(numbers)-1): #0, 1, 2, 3 (Iteration 4)

     for idx2 in range(0, len(numbers)- idx1 - 1):
     #5-0-1=4 , 5-1-1=3 , 5-2-1=2, 5-3-1=1

          if (numbers[idx2] > numbers[idx2 + 1]):
               #Swapping places
               temp = numbers[idx2]
               numbers[idx2] = numbers[idx2 + 1]
               numbers[idx2 + 1] = temp

print("After Sorting:", numbers)
```

For simulation(visualization) of the algorithms use this link below.
Link: https://visualgo.net/en/sorting

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/