浙江理工大学
ZHEJIANG SCI-TECH UNIVERSITY

11. Review

《Python programming》 / Lecturer：Zhiyi Luo (罗志一)

School of Computer Science and Technology
计算机科学与技术学院

Data Types

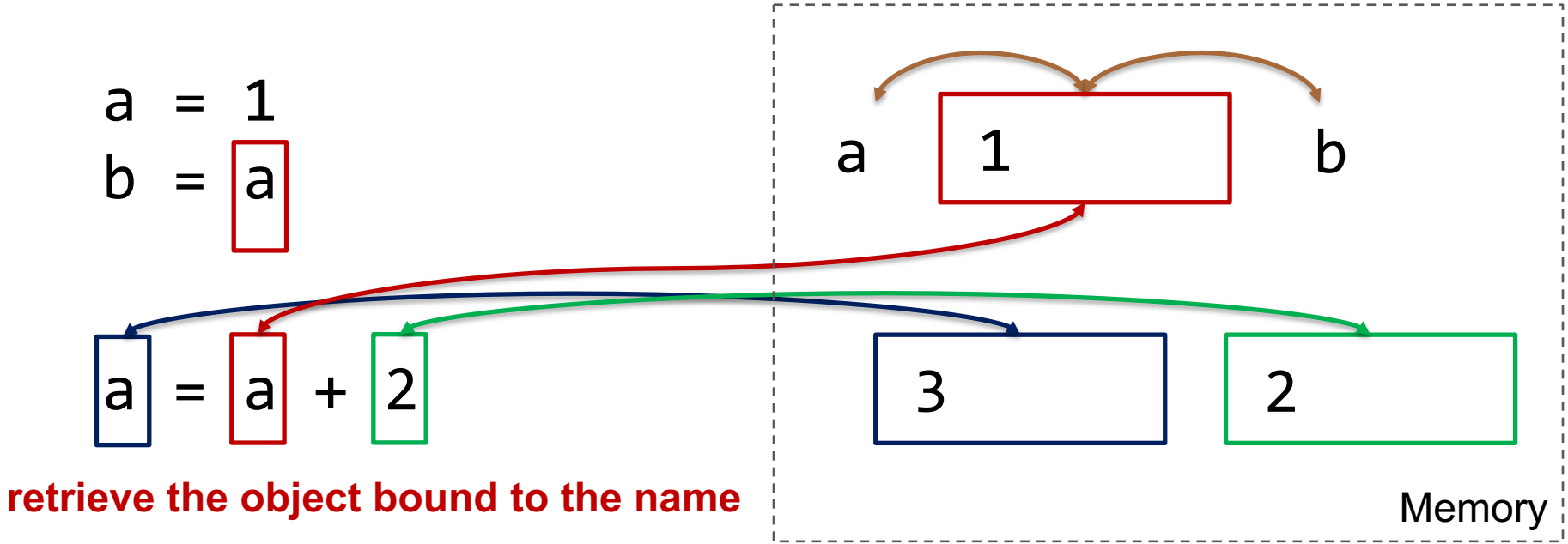
	Type	Mutable	Examples
Numbers	bool	No	True, False
	int	No	13, 256,1024
	float	No	1.21, 3.14, 2e-7
	complex	No	5+9j
String	str	No	“hello”, ‘Jack’
Bytes	bytes	No	b‘ab\xff’
List	list	Yes	[‘Winken’, ’Blinken’, ’Nod’]
Tuple	tuple	No	(2, 4, 8)
Dictionary	dict	Yes	{“name”: “Jack”, “age”: 18}
Set	set	Yes	Set([3, 5, 7])

Mutable & Immutable

- Mutable: can be changed
  - list, dict, set
- Immutable: constant
  - int,float, str, tuple

Modifying an object

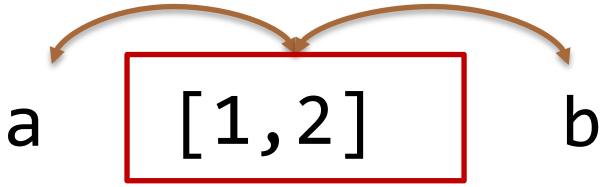
- So far we used immutable objects (floats and integers).



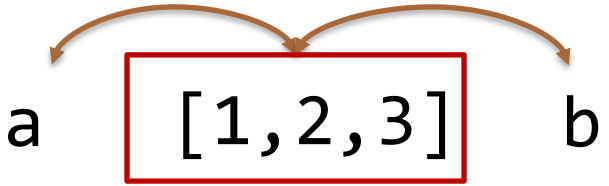
Modifying an object

- So far we used immutable objects (floats and integers). Let's see an example with a list.

```
a = [1, 2]
b = a
```




```
a.append(3)
```



retrieve the object bound to the name

Identifiers

- Identifiers: names given to variables
- The identifier is a combination of character digits and underscore and the character includes letters in lowercase (a-z), letters in uppercase (A-Z), digits (0-9), and an underscore (_).
- An identifier cannot begin with a digit. If an identifier starts with a digit, it will give a Syntax error.
- Special symbols like !, @, #, $, %, etc. are not allowed in identifiers.
- Python identifiers cannot only contain digits.
- There is no restriction on the length of identifiers.
- Identifier names are case-sensitive.

Keywords: Reserved Words

- You cannot use reserved words as variable names / identifiers

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

Identifiers Example

- **Python Valid Identifiers Example**
  - abc123, abc_de, _abc, ABC, abc
- **Python Invalid Identifiers Example**
  - 123abc, abc@, 123, for

Expressions

- An expression is the combination of variables and operators that evaluate based on operator precedence.

Operators

- Arithmetic operators
  - It works the same as basic mathematics.
  - +, -, *, /, // floor division, % modulus, ** exponent

- // floor division
  - 对商向下取整
  - 5//2, 9//4, -7//3, 3.0//2, 3.3//3
- % modulus
  - 取模： a = a%b + b*a//b 对商向下取整
  - 模 a%b 一定是个非负数
  - 8%2, 7%2, -7%3

Operators

- Relational (comparison) operators
  - It performs a comparison between two values. It returns a boolean True or False depending upon the result of the comparison.
  - >, <, ==, !=, >=, <=
- Assignment operators
  - =, +=, -=, *=, /=, %=, //= , **=

Operators

- Logical operators/Boolean operators are useful when checking a condition is true or not. Python has three logical operators. All logical operator returns a boolean value True or False depending on the condition in which it is used.
  - Logical **and**: True if both the operands are True
  - Logical **or**: True if either the operands is True
  - Logical **not**: True if the operand is False

Operators

- Membership operators
  - Python’s membership operators are used to check for membership of objects in sequence, such as string, list, tuple. It checks whether the given value or variable is present in a given sequence. If present, it will return True else False.
  - **in**, **not in**
- Identify operators
  - Use the identify operator to check whether the value of two variables is the same or not. The identify operator compares values according to two variables’ member addresses.
  - **is**, **is not**

Operators

- Bitwise operators
  - In Python, bitwise operators are used to performing bitwise operations on integers. To perform bitwise, we first need to convert integer value to binary (0 and 1) value.
  - The bitwise operator operates on values bit by bit, so it's called **bitwise**. It always returns the result in decimal format. Python has 6 bitwise operators listed below.
  - & bitwise and, | bitwise or, ^ bitwise xor, ~ bitwise 1's complement, << bitwise left-shift, >> bitwise right-shift
  - For example, **&** performs logical AND operation on the integer value after converting an integer to a binary value and gives the result as a demical value.

Operators Precedence

- In Python, operator precedence and associativity play an essential role in solving the expression. We must know what the precedence (priority) of that operator is and how they will evaluate down to a single value.
- Operator precedence is used in an expression to determine which operation to perform first.

Precedence level	Operator	Meaning
1 (Highest)	()	Parenthesis
2	**	Exponent
3	+x, -x ,~x	Unary plus, Unary Minus, Bitwise negation
4	*, /, //, %	Multiplication, Division, Floor division, Modulus
5	+, -	Addition, Subtraction
6	<<, >>	Bitwise shift operator
7	&	Bitwise AND
8	^	Bitwise XOR
9	|	Bitwise OR
10	==, !=, >, >=, <, <=	Comparison
11	is, is not, in, not in	Identity, Membership
12	not	Logical NOT
13	and	Logical AND
14 (Lowest)	or	Logical OR

Lists

- Python list is an ordered sequence of items.
- Ordered: Maintain the orde of the data insertion
- Mutable: List is mutable and we can modify items
- Heterogeneous: List can contain data of different types
- Contains duplicate: Allows duplicates data

L = [20, ‘Jessa’, ‘35.75, [30, 60, 90]]

Lists

- Creating a list
  - Using **list()** constructor
  - Using square bracket **[]**
- Convert other data type to Lists with list()
  - string, tuple -> list
- Get an item by using [offset]
- Get a slice to extract items by offset range
  - Syntax: **lst[start_index: end_index: step]**
  - Slice from left to right when step is positive, otherwise slice from right to left (note that: the slice step can not be zero)

Lists

- Adding elements to the list: append(), insert() and extend()
  - **lst.append(item)**
  - **lst.insert(index, item)**
  - **lst.extend(list_object)**
- Modify the items of a List
- Removing elements from a List
  - **lst.remove(item): to remove the first occurrence of the item from the list `lst`.**
  - **lst.pop(index): Removes and returns the item at the given index from the list `lst`.**

Lists

- Concatenation of two lists
  - Using `+` operator
  - **lst1.extend(lst2)**
- The Pythonic way to check for the existence of a value in a list is using **in**.
- Count occurrences of a value by using count()
  - **lst.count(item): to count how many times the item occurs in the list `lst`.**
  - **lst.pop(index): removes and returns the item at the given index from the list `lst`.**
- Convert to a string with join()

Lists

- Get the length of a list
  - **len(lst)**
- Sort a list using sort()
  - **lst.sort(): to sort the elements in the list `lst` in ascending order.**
- Reverse a list or Get a reversed list
  - **lst.reverse(): reverse the elements in the list `lst`.**
  - **lst[::-1]: returns a new list with reversed elements in the list `lst`.**
- Convert to a string with join()
  - **sep.join(lst): the separator `sep` is a string, and all elements in `lst` should be string.**

Lists

- max(lst)
- min(lst)
- sum(lst)

Tuples

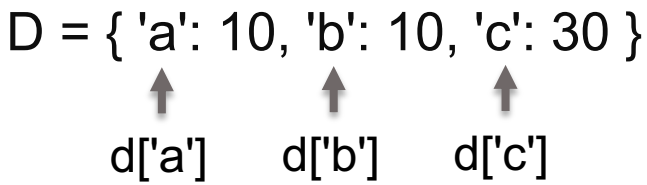
- Creating a tuple
  - Using **tuple()** constructor
  - Using parenthesis **()**
- Packing variables into tuple
  - **tuple1 = 1, 2, "Hello"**
- Unpacking tuple into variables
  - **i, j, k = tuple1**

String

- Chop a string into a list with split()
- Format string

Dictionaries

- Unordered collections of unique values stored in (Key-Value) pairs.



- Unordered: Not reliable, relevant to the item insert order.
- Unique: Each value has a key; the keys in Dictionaries should be unique.
- Mutable: The dictionaries are collections that are changeable, which implies that we can add or remove items after the creation.

Creating a dictionary

- Using curly brackets **{Key1:Value1, Key2:Value2, …}**
  - The dictionaries are create by enclosing the comma-separated **Key:Value** pairs inside the **{}** curly brackets. The colon **:** is used to separate the key and value in a pair.
- Using **dict()** constructor
  - Create a dictionary by passing the comma-separated **Key:Value** pairs inside the dict().
- Convert by using dict()
  - Using **dict()** to convert two-value sequences into a dictionary. The first item in each sequence is used as the key and the second as the value.

Dictionaries

- Get an item by [key]
  - **d[key]**
  - **d.get(key)** **OR** **d.get(key, default_value)**
- Add or Change an item by [key]
  - **d[key] = value**
- Get all keys by using keys()
  - **d.keys()**
- Get all values by using values()
  - **d.keys()**
- Test for a key by using **in**
  - **key in d**
  - **key in d.keys()**

Dictionaries

- Get an item by [key]
  - **d[key]**
  - **d.get(key)** **OR** **d.get(key, default_value)**
- Add or Change an item by [key]
  - **d[key] = value**
- Get all keys by using keys()
  - **d.keys()**
- Get all values by using values()
  - **d.values()**
- Get all key-value pairs by using items()
  - **d.items()**

Sets

S = { 20, ‘Jessa’, 35.75, [20, 30, 40] }

- Unordered: Set doesn’t maintain the order of the data insertion. The items will be in different order each time when we access the Set object. There will not be any index value assigned to each item in the set.
- Immutable: Set are immutable and we can’t modify items. Set items must be immutable. We cannot change the set items, i.e., We cannot modify the items’ value. But we can add or remove items to the Set. A set itself may be modified, but the elements contained in the set must be of an immutable type.

Sets

S = { 20, ‘Jessa’, 35.75 }

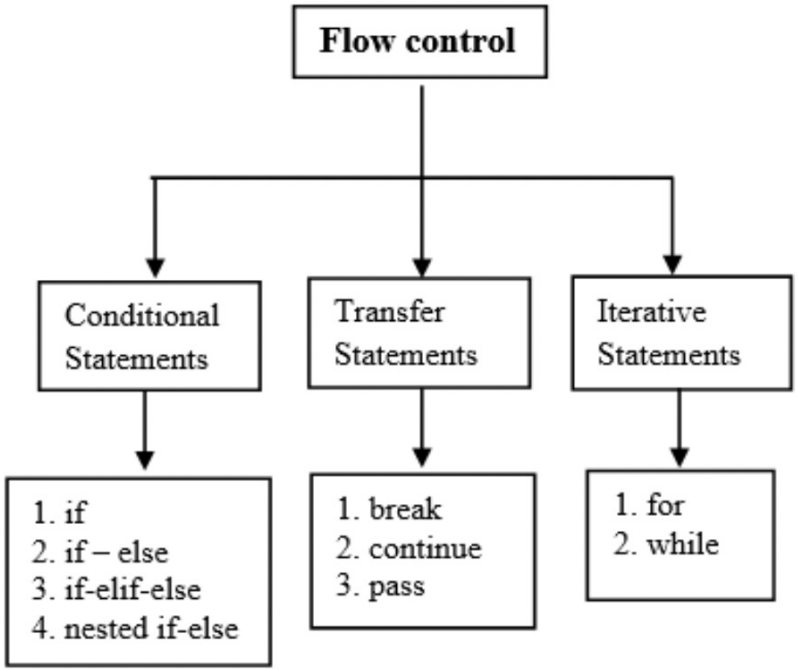
- Heterogeneous: Set can contains data of all types (immutable).
- Unique: Set doesn’t allows duplicates items

Sets

- Create with curly bracket **{item1, item2, …}**
- Create with **set()**
- Convert from Other Data Types with set()
  - set('letters')
  - set(['Dasher', 'Dancer', 'Prancer', 'Mason-Dixon'])
  - set({'apple': 'red', 'orange': 'orange', 'cherry': 'red'})
- Set operators
  - &, |, -
  - >, >=, <, <=

Control Flow Statements

- Conditional statements
- Iterative statements
- Transfer statements



Change the normal flow of execution

Functions

```
def func(a, b):
    ...
    return ...

func(1, 2)
func(var1, var2)
return_value = func(3, 4)
```

函数定义

- 函数定义将新定义的函数对象绑定到选定的名称上
- 函数是一种复合语句：
  - 函数内部的语句在调用时执行，每次调用执行一次
- 函数有额外的名称：参数
  - 参数名称对应的值，在调用时，由调用者进行绑定
- 函数可以有返回值，返回值是调用函数的表达式的值
  - 用return语句返回值
  - 没有执行return语句就返回了，或者return后面没有跟表达式的情况下，返回值是None

```
def add(a,b):
    return a + b
```

参数绑定

- 调用方有两种绑定参数的方式：
  - 按顺序绑定：从左到右指定的每个参数，绑定到参数列表中从左到右对应的名称上
  - 按名称绑定：使用 name = value的方式，直接指定要绑定的参数
- 两种方式可以混用，但不可以同时指定同一个参数
- 参数在函数内可以重新绑定到其他值，解开与调用方传入值的关系

```
def func(a,b):
    ...
func(1,2)
func(a = 1, b = 2)
func(1, b = 2)
func(1, 2, a = 3) # 出错
```

默认参数

- 定义函数时，可以指定参数的默认值，如果调用方未绑定该参数，则使用默认值
  - 可以理解为：在定义函数时已经绑定了该参数的值，如果调用时重新指定则重新绑定该参数到新的值，否则使用旧的绑定
  - 服从绑定的一般规则：所有调用中默认参数绑定的是相同对象
  - 默认参数的绑定在定义时进行

警惕：绑定默认值到可变类型时需要格外注意

```
def func(v, a = []):
    a.append(v)
    return a
a = func(1)       # [1]
b = func(1)       # [1,1]
a  #  [1,1]
```

- 默认值必须从后向前指定，不能在有默认值的参数后加没有默认值的参数

可变参数

- 可变参数是特殊的参数绑定方式

```
def var_func(a, b, *more_args, **more_kw_args):
    ...
var_func(1,2,3,4, test = 5)  # a = 1, b = 2, more_args = (3,4),
                             # more_kwargs = {'test': 5}
```

- 如果定义了*args形式的参数，调用时，所有没有匹配到其他参数的多余的位置参数，构成元组，绑定到args
- 如果定义了**kwargs形式的参数，调用时，所有没有匹配到其他参数的多余的命名参数，构成字典，绑定到kwargs

可变参数的相反操作：解构绑定参数

调用时可以使用元组和字典提供参数

```
def func(a,b):
    ...
c = (1,2)
func(*c) # 等效于：func(1,2)
d = {'a': 1, 'b': 2}
func(**d) # 等效于：func(a = 1, b = 2)
```

Scope and Lifetime of Variables

- When we are executing a function, the life of the variables is up to running time. Once we return from the function, those variables get destroyed. So function does no need to remember the value of a variable from its previous call.

Local Variable in function

- A local variable is a variable declared inside the function that is not accessible from outside of the function. The scope of the local variable is limited to that function only where it is declared.
- If we try to access the local variable from the outside of the function, we will get the error as NameError.

Local Variable in function

```
def function1():
    # local variable
    loc_var = 888
    print("Value is :", loc_var)

def function2():

    print("Value is :", loc_var)

function1()
function2()
```

Value is : 888 print("Value is :", loc_var) # gives error, NameError: name 'loc_var' is not defined

Global Variable in function

- In Python, global is the keyword used to access the actual global variable from outside the function. We use the global keyword for two purposes:
  - To declare a global variable inside the function
  - Declaring a variable as global, which makes it available to function to perform the modification

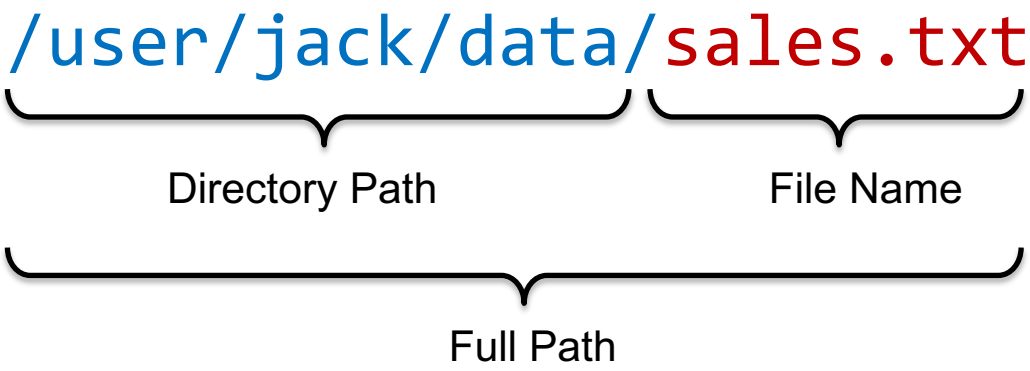
Global Variable in function

```
x = 5  # Global variable

# defining 1st function
def function1():
    print("Value in 1st function :", x)

# defining 2nd function
def function2():
    # Modify global variable using global keyword
    global x
    x = 555
    print("Value in 2nd function :", x)

# defining 3rd function
def function3():
    print("Value in 3rd function :", x)

function1()
function2()
function3()
```

Files

- A file path define the location of a file or folder in the computer system. There are two ways to specify a file path.
  - **Absolute path**: which always begins with the root folder
  - **Relative path**: which is relative to the program's current working directory

Absolute Path



Read or Write Files

- `f = open("sample.txt", "r")`
- `f = open("sample.txt", "w")`

File Access Modes

Mode	Description
r	It opens an existing file to read-only mode. The file pointer exists at the beginning.
rb	It opens the file to read-only in binary format. The file pointer exists at the beginning.
r+	It opens the file to read and write both. The file pointer exists at the beginning.
rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name.
wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists.
w+	It opens the file to write and read data. It will override existing data.
wb+	It opens the file to write and read both in binary format
a	It opens the file in the append mode. It will not override existing data. It creates a new file if no file exists with the same name.
ab	It opens the file in the append mode in binary format.
a+	It opens a file to append and read both.
ab+	It opens a file to append and read both in binary format.

File Methods

- There are various methods available that we can use with the file object.

Method	Description
read([size])	Returns the file content.
readline()	Read single line 逐行读取文件
readlines()	Read file into a list 读取文件中的所有行
write()	Writes the specified string to the file.
writelines()	Writes a list of strings to the file.
close()	Closes the opened file.
seek()	Set file pointer position in a file
tell()	Returns the current file location.
flush()	Flushes the internal buffer.

Scanning File Content

```
1  for line in open("test.txt").readlines():
2      print(line, end='')
3
4  for line in open("test.txt"):
5      print(line, end='')
6
7  with open("test.txt") as f:
8      for line in f:
9          print(line, end='')
```

Object-oriented Programming

- **Objected-oriented programming (OOP**) is a programming paradigm based on the concept of **“objects”.** The object contains both data and code; Data in the form of properties (often known as attributes), and code, in the form of methods (actions object can perform).
- An object-oriented paradigm is to design the program using classes and objects. An object has the following two characteristics:
  - Attribute
  - Behavior
- One important aspect of OOP in Python is to create reusable code using the concept of inheritance. This concept is also known as DRY (Don’t Repeat Yourself).

Class and Objects

- Class: The class is a user-defined data structure that binds the data members and methods into a single unit. Class is a blueprint or code template for object creation. Using a class, you can create as many objects as you want.
- Object: An object is an instance of a class. It is a collection of attributes (variable) and methods. We use the object of a class to perform actions.
- Objects have two characteristics: They have states and behaviors (object has attributes and methods attached to it) Attributes represent its state, and methods represent its behavior. Using its methods, we can modify its state.

```
class Employee:
    # class variables
    company_name = 'ABC Company'
    # constructor to initialize the object
    def __init__(self, name, salary):
        # instance variables
        self.name = name
        self.salary = salary
    # instance method
    def show(self):
        print('Employee:', self.name, self.salary, self.company_name)

# create first object
emp1 = Employee("Harry", 12000)
emp1.show()

# create second object
emp2 = Employee("Emma", 10000)
emp2.show()
```

Example Explanation

- In the above example, we create a Class with the name Employee.
- Next, we defined two attributes name and salary.
- Next, in the __init__() method, we initialized the value of attributes. This method is called as soon as the object is created. The init method initializes the object.
- Finally, from the Employee class, we created two objects, Emma and Harry.
- Using the object, we can access and modify its attributes.

```
class Employee:
    # class variables
    company_name = 'ABC Company'
    # constructor to initialize the object
    def __init__(self, name, salary):
        # instance variables
        self.name = name
        self.salary = salary
    # instance method
    def show(self):
        print('Employee:', self.name, self.salary, self.company_name)

# create first object
emp1 = Employee("Harry", 12000)
emp1.show()

# create second object
emp2 = Employee("Emma", 10000)
emp2.show()
```

constructor

Parameters to constructor

Instance variable

Instance method

Object of class