PYTHON CƠ BẢN (Buổi 3)

AI Academy Vietnam

Content

I. Python Introduction

II. Python Flow Control

III. Python Functions

IV. Python Datatypes

V. Python Files

VI. Python Object & Class

VII. Python Date & Time

IV. Python Datatypes

1. Numbers
2. String
3. Collection datatypes

IV.1. Numbers

- Python supports integers, floating-point numbers and complex numbers.
- They are defined as **`int`**, **`float`**, and **`complex`** classes.
- While integers can be of any length, a floating-point number is accurate only up to 15 decimal places (the 16th place is inaccurate).

IV.1. Numbers

- Number objects are created when you assign a value to them

```
var1 = 1
var2 = 10
```

- Changing the value of a number data type results in a newly allocated object.
- Delete the reference to a number object by using the **del** statement.

```
del var1[,var2[,var3[....,varN]]]
```

IV.1. Numbers

- Numeric Operators

Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	floored quotient of *x* and *y*
`x % y`	remainder of `x / y`
`-x`	*x* negated
`+x`	*x* unchanged
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`float(x)`	*x* converted to floating point
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
`c.conjugate()`	conjugate of the complex number *c*
`divmod(x, y)`	the pair `(x // y, x % y)`
`pow(x, y)`	*x* to the power *y*
`x ** y`	*x* to the power *y*

IV.1. Numbers

- Mathematics Function
  - Module `math`: https://docs.python.org/3.6/library/math.html
  - Some methods are provided by this module
    - `ceil(x), copysign(x,y), fabs(x),factorial(x), floor(x), exp(x), log(x), cos(x), acos(x), radians(x)`
  - Constants: `math.pi, math.e, math.inf, math.tau, ..`
  - Example:

```
import math
print(math.sin(math.pi))
```

IV.2. Strings

- String literals in python are surrounded by either single quotation marks, or double quotation marks.
- Display a string literal with the `print()` function

```
print("Hello")
print('Hello’)
```

- Assign String to a Variable

```
x = "Hello"
```

- Multiline Strings

```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
```

IV.2. Strings

- Common Sequence Operations

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

IV.2. Strings

- Escape characters

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

IV.2. Strings

- Build-in methods
  - `capitalize, casefold, center, split, count, islower, isupper, encode, find,…`
  - **See more:** https://docs.python.org/3/library/stdtypes.html#string-methods
  - All string methods returns new values. They do not change the original string.

IV.2. Strings

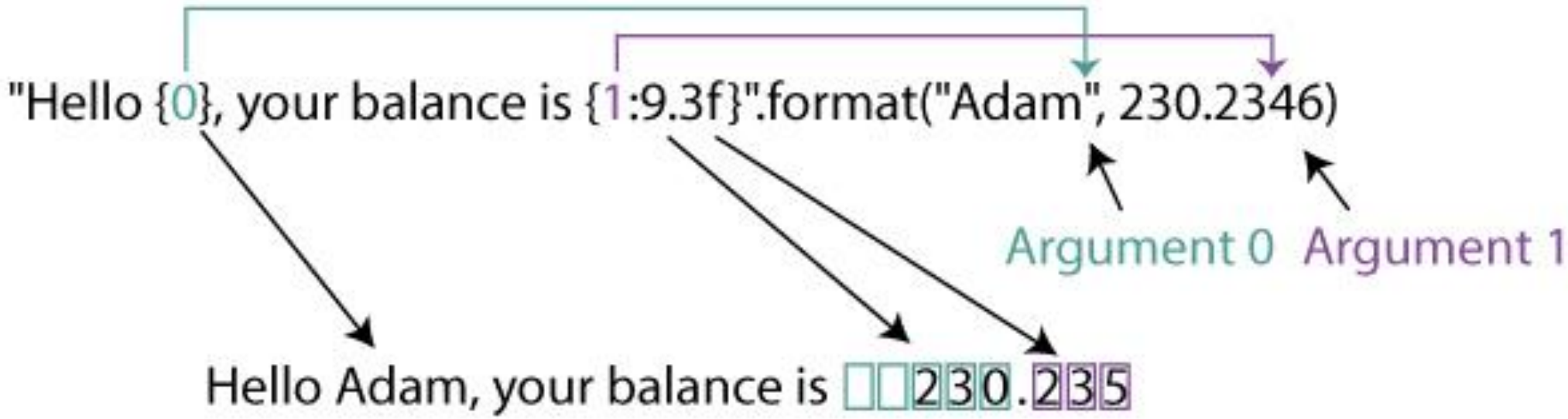
- String formatting operator
  - Syntax of format():

    `template.format(p0, p1, ..., k0=v0, k1=v1, ...)`

    - `p0, p1,...` are positional arguments
    - `k0, k1,...` are keyword arguments with values v0, v1,...
    - `template` is a mixture of format codes with placeholders for the arguments.
  - Positional parameters - list of parameters that can be accessed with index of parameter inside curly braces {index}
  - Keyword parameters - list of parameters of type key=value, that can be accessed with key of parameter inside curly braces {key}

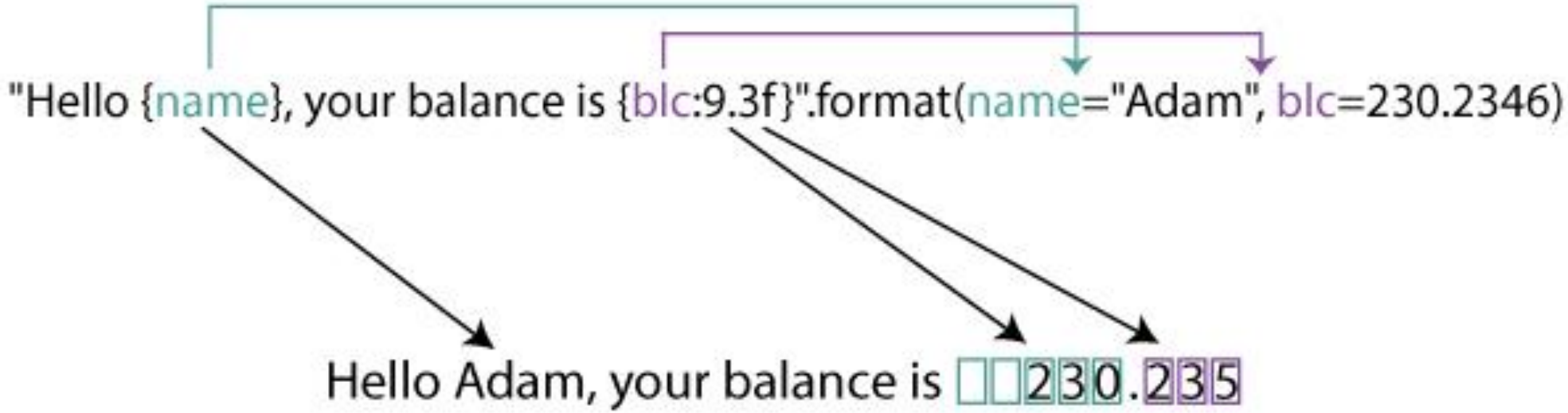
IV.2. Strings

- String formatting operator
  - **For positional arguments**



IV.2. Strings

- String formatting operator
  - **For keyword arguments**



IV.2. Strings

- String formatting operator
  - **Numbers formatting with format**

Type	Meaning
d	Decimal integer
c	Corresponding Unicode character
b	Binary format
o	Octal format
x	Hexadecimal format (lower case)
X	Hexadecimal format (upper case)
n	Same as 'd'. Except it uses current locale setting for number separator

IV.2. Strings

- String formatting operator
  - **Numbers formatting with format (cont.)**

Type	Meaning
e	Exponential notation. (lowercase e)
E	Exponential notation (uppercase E)
f	Displays fixed point number (Default: 6)
F	Same as 'f'. Except displays 'inf' as 'INF' and 'nan' as 'NAN'
g	General format. Rounds number to p significant digits. (Default precision: 6)
G	Same as 'g'. Except switches to 'E' if the number is large.
%	Percentage. Multiples by 100 and puts % at the end.

IV.2. Strings

- Example:

```
# integer arguments
print("The number is:{:d}".format(123))

# float arguments
print("The float number is:{:f}".format(123.4567898))

# octal, binary and hexadecimal format
print("bin: {0:b}, oct: {0:o}, hex: {0:x}".format(12))
```

```
The number is:123
The float number is:123.456790
bin: 1100, oct: 14, hex: c
```

IV.2. Strings

- Example:

```
# integer numbers with minimum width
print("{:5d}".format(12))

# width doesn't work for numbers longer than padding
print("{:2d}".format(1234))

# padding for float numbers
print("{:8.3f}".format(12.2346))

# integer numbers with minimum width filled with zeros
print("{:05d}".format(12))

# padding for float numbers filled with zeros
print("{:08.3f}".format(12.2346))
```

```
   12
1234
  12.235
00012
00012.235
```

IV.2. Strings

- Number formatting with alignment

Type	Meaning
<	Left aligned to the remaining space
^	Center aligned to the remaining space
>	Right aligned to the remaining space
=	Forces the signed (+) (-) to the leftmost position

IV.2. Strings

- Example:

```
# integer numbers with right alignment
print("{:5d}".format(12))

# float numbers with center alignment
print("{:^10.3f}".format(12.2346))

# integer left alignment filled with zeros
print("{:<05d}".format(12))

# float numbers with center alignment
print("{:=10.3f}".format(12.2346))
```

```
   12
  12.235
12000
-  12.235
```

IV.3. Collection data types

- There are four collection data types in the Python programming language:
  - **List** is a collection which is ordered and changeable. Allows duplicate members.
  - **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
  - **Set** is a collection which is unordered and unindexed. No duplicate members.
  - **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

IV.3. Collection data types

- Common methods

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

IV.3. Collection data types

- List
  - A list is created by placing all the items (elements) inside a square bracket [ ], separated by commas.
  - It can have any number of items and they may be of different types (integer, float, string etc.).

```
# empty list
my_list = []
# list of integers
my_list = [1, 2, 3]
# list with mixed datatypes
my_list = [1, "Hello", 3.4]
```

  - A list can even have another list as an item
  - The index operator [] is used to access an item in a list

IV.3. Collection data types

- List methods

append() - Add an element to the end of the list

extend() - Add all elements of a list to the another list

insert() - Insert an item at the defined index

remove() - Removes an item from the list

pop() - Removes and returns an element at the given index

clear() - Removes all items from the list

index() - Returns the index of the first matched item

count() - Returns the count of the number of items passed as an argument

sort() - Sort items in a list in ascending order

reverse() - Reverse the order of items in the list

copy() - Returns a shallow copy of the list

IV.3. Collection data types

- List methods

```
# Python list methods
my_list = [3, 8, 1, 6, 0, 8, 4]

# Output: 1
print(my_list.index(8))

# Output: 2
print(my_list.count(8))

my_list.sort()

# Output: [0, 1, 3, 4, 6, 8, 8]
print(my_list)

my_list.reverse()

# Output: [8, 8, 6, 4, 3, 1, 0]
print(my_list)
```

```
1
2
0, 1, 3, 4, 6, 8, 8]
8, 8, 6, 4, 3, 1, 0]
```

IV.3. Collection data types

- Tuple
  - A tuple is created by placing all the items (elements) inside parentheses (), separated by commas.
  - A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Empty tuple
my_tuple = ()
# Tuple having integers
my_tuple = (1, 2, 3)
# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
```

IV.3. Collection data types

- Tuple:
  - A tuple can also be created without using parentheses. This is known as tuple packing.

```
my_tuple = 3, 4.6, "dog"
# tuple unpacking is also possible
a, b, c = my_tuple
```

  - Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

```
my_tuple = ("Hello", )
```

IV.3. Collection data types

- Tuple
  - Once a tuple is created, you cannot change its values.

```
x = ("apple", "banana", "cherry")
x[1] = "kiwi"
```

  - You can convert the tuple into a list, change the list, and convert the list back into a tuple.

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

```
('apple', 'kiwi', 'cherry')
```

IV.3. Collection data types

- **Advantages of Tuple over List**
  - We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
  - Since tuples are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
  - Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
  - If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

IV.3. Collection data types

- Set:
  - A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function set().
  - It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element.
  - Cannot access or change an element of set using indexing or slicing.

```
# set of integers
my_set = {1, 2, 3}
# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
# empty set
my_set = set()
```

IV.3. Collection data types

- Set methods:

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another

IV.3. Collection data types

- Set methods:

Method	Description
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
pop()	Removes and returns an arbitrary set element. Raises KeyError if the set is empty
remove()	Removes an element from the set. If the element is not a member, raises a KeyError
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

IV.3. Collection data types

- Set methods:

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# Intersection of sets
# use & operator
print(A & B)

# use intersection function on A
print(A.intersection(B))

# Union of sets
# use | operator
print(A | B)

# use union function
print(A.union(B))
```

```
{4, 5}
{4, 5}
{1, 2, 3, 4, 5, 6, 7, 8}
{1, 2, 3, 4, 5, 6, 7, 8}
```

IV.3. Collection data types

- Dictionary:
  - While other compound data types have only value as an element, a dictionary has a key: value pair.
  - Dictionaries are optimized to retrieve values when the key is known.
  - Creating a dictionary is as simple as placing items inside curly braces {} separated by comma.
  - While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

IV.3. Collection data types

- Example

```
# empty dictionary
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict =dict( {'name': 'John', 1: [2, 4, 3]})

# from sequence having each item as a pair
my_dict = dict([(1, 'apple'), (2, 'ball')])
```

IV.3. Collection data types

- Dictionary:
  - Access element in a dictionary:

```
my_dict = {'name':'Jack', 'age': 26}
# Output: Jack
print(my_dict['name’])
# Output: 26
print(my_dict.get('age’))
```

  - Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.
  - If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

IV.3. Collection data types

- Dictionary methods:

Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Returns the value of the key. If the key does not exist, returns d (defaults to None).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.
pop(key[,d])	Removes the item with the key and returns its value or d if key is not found. If d is not provided and the key is not found, it raises KeyError.

IV.3. Collection data types

- Dictionary methods:

Method	Description
popitem()	Removes and returns an arbitrary item (**key, value**). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the key is in the dictionary. If not, inserts the key with a value of d and returns d (defaults to None).
update([other])	Updates the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Returns a new object of the dictionary's values

VI. Object Oriented Programming

- Class
- Object
- Method
- Inheritance
- Encapsulation
- Polymorphism

VI. Object Oriented Programming

- Python is a multi-paradigm programming language. It supports different programming approaches.
- One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).
- An object has two characteristics:
  - Attributes
  - Behavior
- The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

VI. Object Oriented Programming

- A **class** is a blueprint for the object.

```
class Parrot:
        pass
```

- From class, we construct instances. An instance is a specific object created from a particular class.
- An **object** (instance) is an instantiation of a class. When class is defined, only the description for the object is defined.

```
obj = Parrot()
```

VI. Object Oriented Programming

- Example

```
class Parrot:
    # class attribute
    species = "bird"
    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age
# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)
# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))
# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

VI. Object Oriented Programming

- **Methods** are functions defined inside the body of a class.

```
Blu sings 'Happy'
Blu is now dancing
```

```
class Parrot:
    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)
    def dance(self):
        return "{} is now dancing".format(self.name)
# instantiate the object
blu = Parrot("Blu", 10)
# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

VI. Object Oriented Programming

- **Inheritance** is a way of creating a new class for using details of an existing class without modifying it.
- The newly formed class is a derived class (or child class).
- Similarly, the existing class is a base class (or parent class).

VI. Object Oriented Programming

```
# parent class
class Bird:
    def __init__(self):
        print("Bird is ready")
    def whoisThis(self):
        print("Bird")
    def swim(self):
        print("Swim faster")
```

```
# child class
class Penguin(Bird):
    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")
    def whoisThis(self):
        print("Penguin")
    def run(self):
        print("Run faster")
peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

VI. Object Oriented Programming

- Using OOP in Python, we can restrict access to methods and variables.
- This prevents data from direct modification which is called **encapsulation**.
- In Python, we denote private attributes using underscore as the prefix i.e single _ or double __.

VI. Object Oriented Programming

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))
    def setMaxPrice(self, price):
        self.__maxprice = price
```

```
c = Computer()
c.sell()
# change the price
c.__maxprice = 1000
c.sell()
# using setter function
c.setMaxPrice(1000)
c.sell()
```

```
Selling Price: 900
Selling Price: 900
Selling Price:1000
```

VI. Object Oriented Programming

- **Polymorphism** is an ability (in OOP) to use a common interface for multiple forms (data types).

```
class Parrot:
    def fly(self):
        print("Parrot can fly")
     def swim(self):
        print("Parrot can't swim")
```

```
class Penguin:
    def fly(self):
        print("Penguin can't fly")
    def swim(self):
        print("Penguin can swim")
```

```
# common interface
def flying_test(bird):
    bird.fly()
#instantiate objects
blu = Parrot()
peggy = Penguin()
# passing the object
flying_test(blu)
flying_test(peggy)
```

```
Parrot can fly
Penguin can't fly
```

Content

I. Python Introduction
II. Python Flow Control
III. Python Functions
IV. Python Datatypes
V. Python Files : https://www.programiz.com/python-programming/file-operation
VI. Python Object & Class
VII. Python Date & Time : https://www.programiz.com/python-programming/datetime