Python

Python is a **high-level** programming language which is:

Interpreted: Python is processed at run time by the interpreter.

Interactive: You can use a Python prompt and interact with the interpreter directly to write your programs.

Object-Oriented: Python supports Object-Oriented technique of programming.

Beginner's Language: Python is a great language for the beginner-level programmers and supports the development of a wide range of applications.

History

Python was conceptualized by **Guido Van Rossum** in the late **1980s**.

Rossum published the first version of Python code (0.9.0) in February **1991** at the CWI (Centrum Wiskunde & Informatica) in the Netherlands , Amsterdam.

Python is derived from **ABC** programming language, which is a general-purpose programming language that had been developed at the CWI.

Rossum chose the name **"Python"**, since he was a big fan of Monty Python's Flying Circus.

Python is now maintained by a core development team at the institute, although Rossum still holds a vital role in directing its progress.

Features

- Python is **Easy** to learn, easy to read and easy to maintain.

- Python is **Portable**, It can run on various hardware platforms and has the same interface on all platforms.

- Python is **Extendable**, You can add low-level modules to the Python interpreter.

- Python is **Scalable**, Python provides a good structure and support for large programs.

- Python has a **broad standard library** cross-platform.

- **Everything in Python is an object**, variables, functions, even code. Every Object has an ID, a type, and a value.

```
>>> x=36
>>> id(x)
4297539008
>>> type(x)
<class 'int'>
```

- Python provides interfaces to all major commercial **databases**.
- Python supports **functional** and **structured** programming methods as well as **Object Oriented Programming**.
- Python provides very high-level **dynamic data types** and supports **dynamic type checking**.
- Python supports **GUI** applications
- Python supports **automatic garbage collection**.
- Python can be easily **integrated** with C, C++, and Java.

Versions

Python 1.0 (January 1994)
latest minor version is 1.6

this version is discontinued.

Python 2.0 (October 2000)
latest minor version is 2.7

this version will be discontinued in 2020.

Python 3.0 (December 2008)
latest minor version is 3.7

this is the current version of python.

Python 2 vs Python 3

print statement has been replaced with **print()** function.

```
# python 2
print "Hello World!"
# python 3
print("Hello World!")
```

There is only one integer type left, **int**.

Some methods such as **map()** and **filter()** return iterator objects in Python 3 instead of lists in Python 2.

In Python 3, a **TypeError** is raised as warning if we try to compare unorderable types. e.g. **0 > None** is no longer valid.

Python 3 provides Unicode (utf-8) strings while Python 2 has ASCII **str()** types and separate **unicode()**.

A new built-in string formatting method **format()** replaces the **%** string formatting operator.

In Python 3, we should **enclose** the **exception argument** in **parentheses**.

```
# python 2
raise IOError, "file error"
# python 3
raise IOError("file error")
```

In Python 3, we have to use the **as** keyword now in the handling of **exceptions**.

```
# python 2
Try:
    ...
except NameError, err:
    ...

# python 3
Try:
    ...
except NameError as err:
    ...
```

The **division** of two integers returns a **float** instead of an **integer**. **//** can be used to have the old behavior.

Installation

Windows: download installer from following address

```
https://www.python.org/downloads/windows/
```

(remember to select **‘Add Python too PATH’** while installation)

Ubuntu/Debian:

```
~ sudo apt install python3
```

Fedora:

```
~ sudo yum install python3
```

Arch/Manjaro:

```
~ sudo pacman -S python
```

MacOS:

```
~ brew install python
```

for **source files** and **packages** for almost any operating systems check following address:

```
https://www.python.org/downloads/
```

Python must be added to OS PATH variable to be callable.

In **Ubuntu** **python** command used for **Python 2 version** and **python3** is used for **Python 3 version**. it could be different in each OS

Checking installed **Python version**

```
~ Python3 --version
```

Running Python 3 **shell**:

```
~ python3
```

Running Python 3 **script**:

```
~ python3 /path/to/script_file.py
```

PIP Package Manager

Windows: Python installer will install pip too.

```
https://www.python.org/downloads/windows/
```

Ubuntu/Debian:

```
~ sudo apt install python3-pip
```

Fedora:

```
~ sudo yum install python3
```

Arch/Manjaro:

```
~ sudo pacman -S python-pip
```

MacOS: Installing Python with brew will install pip too.

```
~ brew install python
```

Using **pip** too **install** a package: (**pip3** command could be **pip** in some OSes like Windows, in Ubuntu it is pip3)

```
~ pip3 install package-name
```

uninstalling a package:

```
~ pip3 uninstall package-name
```

Basic Syntax

Python files have **.py** extension.

Indentation is used in Python to delimit blocks. The number of spaces is variable, but all statements within **the same block** must be **indented the same amount**.

The header line for compound statements, such as **if**, **while**, **def**, and **class** should be terminated with a **colon (:)**.

The **semicolon (;)** is optional at the end of statement, but it is **preferred to not using it**.

```
if True:
    print("Answer")
    print("True")
else:
    print("Answer")
 print("False") # Error! Not using same indention.
```

Printing to the Screen

```python
print("Hello World!")
```

Reading Keyboard Input

```python
name = input("Enter your name: ")
```

Comments

```python
# one line comment in python!

"""
Multi line comments in in python,
Is like this!
"""
```

Python is **dynamically typed**. You do not need to declare variables!

The **declaration happens automatically** when you assign a value to a variable.

Variables can change type, simply by assigning them a new value of a different type.

```
counter = 1000
miles = 1000.0
name = "Abolfazl"
x = None
x = 2
x = "string"
```

Python allows you to assign a **single value to several variables** simultaneously, and also allows to assign **multiple values to multiple variables** too.

```
a = b = c = 3
x, y, z = 1, 2, "string"
```

Numbers

Numbers are **Immutable objects** in Python that cannot change their values.

There are three built-in data types for numbers in Python3:

- Integer (**int**)
- Floating-point numbers (**float**)
- **Complex** numbers: <real part> + <imaginary part>j

Common Number Functions

Function	Description
int(x)	to convert **x** to an **integer**
float(x)	to convert **x** to a **floating-point** number
abs(x)	The **absolute** value of **x**
cmp(x, y)	**-1** if **x < y**, **0** if **x == y**, or **1** if **x > y**
exp(x)	The exponential of **x: e x**
log(x)	The **natural logarithm** of **x**, for **x > 0**
pow(x, y)	The value of x**y
sqrt(x)	The square **root** of **x** for **x > 0**

Strings

Python Strings are **Immutable** objects that cannot change their values.

```
>>> str1 = "strings are immutable!"
>>> str1[0] = "S"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

You can update an existing string by **(re)assigning a variable** to another string.

Python does not support a character type; these are treated as strings of length one.

Python accepts single **(')**, double **(")** and triple **(''' or """)** quotes to denote string literals.

```
str1 = 'str1'
str2 = '''str2'''
str3 = "str3"
str4 = """str4"""
```

String indexes starting at **0 in the beginning** of the string and working their way from **-1 at the end**.

```
positive indexes    >>    | 0 | 1 | 2 | 3 | 4 |
string is 'HELLO'   >>    | H | E | L | L | O |
negative indexes    >>    |-5 |-4 |-3 |-2 |-1 |
```

String Formatting:

```
>>> num = 6
>>> string = "I have {} books!".format(num)
>>> print(string)
I have 6 books!
```

Common String Operators, Assume: **a='Hello'** and **b='Python'**

Operator	Description	Example
+	**Concatenation** - Adds values on either side of the operator	a+b >>> HelloPython
*	**Repetition** - Creates new strings, concatenating multiple copies of the same string	a*2 >>> HelloHello
[]	**Slice** - Gives the character from the given index	a[1] >>> e a[-1] >>> o
[:]	**Range Slice** - Gives the characters from the given range	a[1:4] >>> ell
in	**Membership** - Returns true if a character exists in the given string	'H' in a >>> True

Common String Methods

Function	**Description**
str.count(sub, beg=0, end=len(str))	**Counts** how many times sub occurs in string or in a substring of string if starting index beg and ending index end are given.
str.isalpha()	Returns **True** if string has at least **1 character** and all characters are alphanumeric and **False** otherwise.
str.isdigit()	Returns **True** if string contains only digits and False otherwise.
str.lower()	Converts letters in string to **lowercase**.
str.upper()	Converts letters in string to **uppercase**.
str.replace(old, new)	**Replaces** all occurrences of **old** in string with **new**.
str.split(str=‘ ’)	**Splits** string according to delimiter str (space if not provided) and returns **list of substrings**.
str.strip()	**Removes** all leading and trailing **white spaces** of string.
str.title()	Returns **"titlecased"** version of string.

Common String Functions

Function	**Description**
str(x)	to convert **x** to an **String**
len(x)	gives the total **length** of the string

Lists

A list in Python is an **ordered** group of items or elements, and these list elements don't have to be of the same type. Lists are **mutable** objects that can change their values.

List indexes like strings starting at **0** in the **beginning** of the list and working their way from **-1** at the **end**.

Similar to strings, Lists operations include slicing (**[]** and **[:]**), concatenation **(+)**, repetition **(*)**, and membership (**in**).

access, **update** and **delete** list elements is like:

```
>>> list1 = ['programming', 'python', 1996, 2019, 0.5]
>>> print(list1[0])
programming
>>> print(list1[1:4])
['python', 1996, 2019]
>>> list1[2] = 2000
>>> print(list1[2])
2000
>>> del(list1[4])
>>> print(list1)
['programming', 'python', 2000, 2019]
```

Lists can have **sub-lists** as elements and these sub-lists may contain other sub-lists as well.

```
>>> persons = [["Abolfazl", 1996], ["Sarah", 1997]]
>>> name = persons[0][0]
>>> birth = persons[0][1]
>>> print("{} was born on {}".format(name, birth))
Abolfazl was born on 1996
```

Common List Functions

Function	**Description**
cmp(list1, list2)	**Compares** elements of both lists.
len(list)	Gives the total **length** of the list.
max(list)	Returns **item** from the list with **max value**.
min(list)	Returns **item** from the list with **min value**.
list(tuple)	**Converts** a tuple into list.

List **Comprehensions** consists of an **expression** followed by a for **clause**.

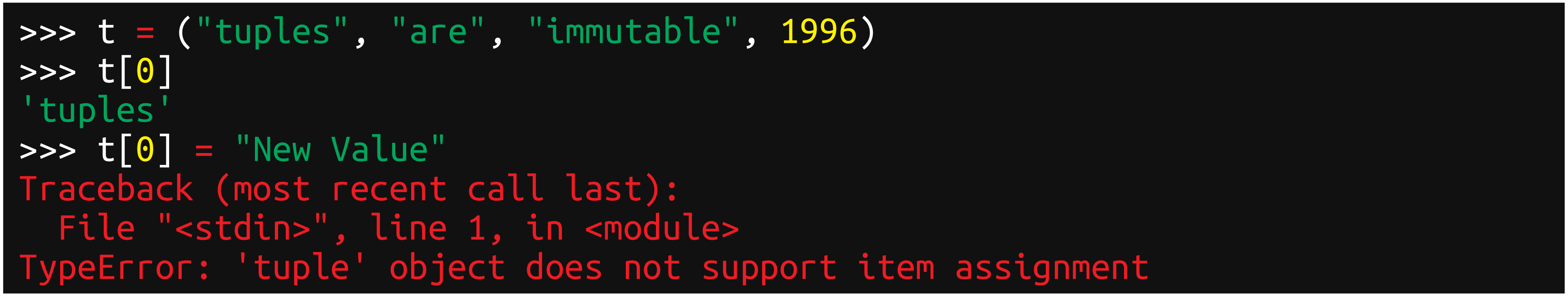
```
>>> a = [1, 2, 3]
>>> [x**2 for x in a]
[1, 4, 9]
>>> [x+1 for x in [x**2 for x in a]]
[2, 5, 10]
```

Common List Methods

Method	**Description**
list.append(obj)	**Appends** object obj to list
list.insert(index, obj)	**Inserts** object obj into list **at offset index**
list.count(obj)	Returns **count** of how many times obj occurs in list
list.index(obj)	Returns the **lowest index** in list that obj appears
list.remove(obj)	**Removes** object obj from list
list.reverse()	**Reverses** objects of list in place
list.sort()	**Sorts** objects of list in place

Tuples

Python Tuples are **Immutable** objects that cannot be changed once they have been created.



```
>>> t = ("tuples", "are", "immutable", 1996)
>>> t[0]
'tuples'
>>> t[0] = "New Value"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

You can **update** an existing tuple by **(re)assigning** a variable to another tuple.

Tuples are **faster** than lists and protect your data against accidental changes to these data.

The rules for tuple indices are the same as for lists and they have the **same operations**, **functions** as well.

To write a tuple containing a **single value**, you have to include a comma, even though there is only one value. e.g. **t = (3,)**

Dictionaries

Python dictionaries are kind of **hash table** type which consist of **key-value** pairs of **unordered** elements.

- **Keys:** must be immutable data types ,usually numbers or strings.
- **Values:** can be any arbitrary Python object.

Python Dictionaries are **mutable** objects that can change their values.

A dictionary is enclosed by **curly braces ({})**, the items are separated by **commas,** and each key is separated from its value by a **colon (:)**.

Dictionary's values can be **assigned** and **accessed** using **square braces ([])** with a key to obtain its value.

Simple script to show dictionary usage:

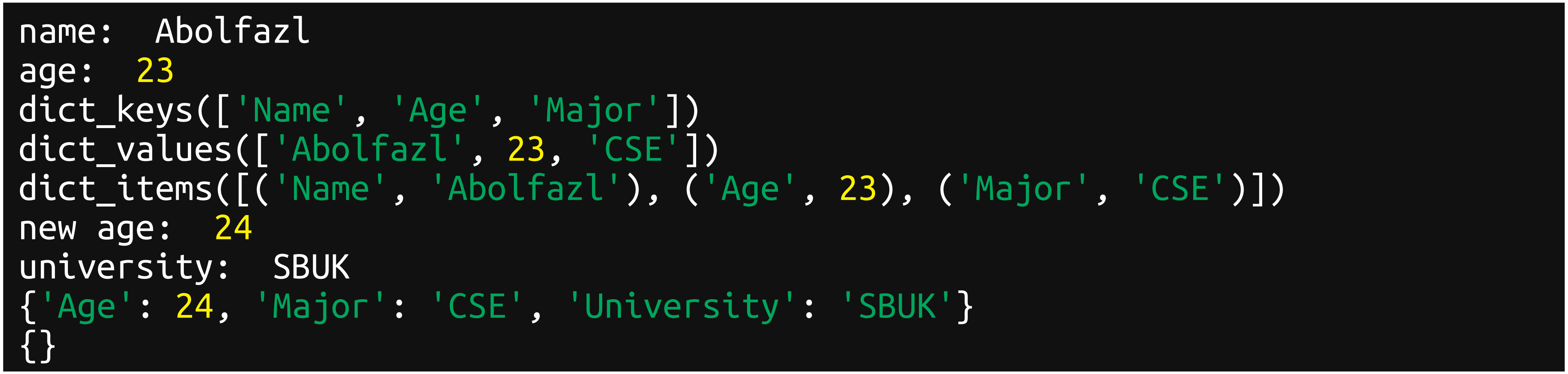
```python
dict1 = {'Name': 'Abolfazl', 'Age':23, 'Major': 'CSE'}

# Access dictionary data
print('name: ', dict1['Name'])
print('age: ', dict1['Age'])
print(dict1.keys())
print(dict1.values())
print(dict1.items())

# Update dictionary data
dict1['Age'] = 24
dict1['University'] = 'SBUK'
print('new age: ', dict1['Age'])
print('university: ', dict1['University'])

# Delete dictionary data
del dict1['Name']
print(dict1)
dict1.clear()
print(dict1)
```

Output:



```
name:  Abolfazl
age:  23
dict_keys(['Name', 'Age', 'Major'])
dict_values(['Abolfazl', 23, 'CSE'])
dict_items([('Name', 'Abolfazl'), ('Age', 23), ('Major', 'CSE')])
new age:  24
university:  SBUK
{'Age': 24, 'Major': 'CSE', 'University': 'SBUK'}
{}
```

Common Dictionary Methods

Function	**Description**
dict.keys()	Returns list of **dict's keys**
dict.values()	Returns list of **dict's values**
dict.items()	Returns a list of **dict's (key, value)** tuple pairs
dict.get(key, default=None)	For **key**, returns **value** or **default** if key not in dict
dict.has_key(key)	Returns **True** if **key in dict**, **False** otherwise
dict.update(dict2)	**Adds** dict2's key-values pairs to dict
dict.clear()	**Removes** all elements of dict

Common Dictionary Functions

Function	**Description**
cmp(dict1, dict2)	**compares** elements of both dict.
len(dict)	gives the **total number** of **(key, value)** pairs in the dictionary.

Conditionals

In Python, **True** and **False** are Boolean objects of class **'bool'** and they are **immutable**.

Python assumes any **non-zero** and **non-null** values as **True**, otherwise it is **False** value.

Python does not provide **switch/case** statements as in other languages.

Example of Python **if** statement

```python
x = int(input("Please enter positive integer: "))
if x < 0:
    x = 0
    print("negative integer changed to zero")
elif x == 0:
    print("zero")
elif x == 1:
    print("single")
else:
    print("multiple")
```

Inline conditional expression

```python
x = "Smaller" if a < b else "Bigger"
```

Loops

for and while loops

```python
for letter in 'Python':
    print(letter, end='-')
print() # end of 1st example
list_data = ['P','y','t','h','o','n']
for d in list_data:
    print(d, end='*')
print() # end of 2nd example
for index in range(len(list_data)):
    print(list_data[index], end='_')
print() # end of 3rd example
count, string = 0, 'Python'
while count <= 5:
    print(string[count], end=' ')
    count+=1
print() # end of 4th example
dict_data = {0:'P', 1:'y', 2:'t', 3:'h', 4:'o', 5:'n'}
for key, value in dict_data.items():
    print(key, value, end=' | ')
```

Output

```
P-y-t-h-o-n-
P*y*t*h*o*n*
P_y_t_h_o_n_
P y t h o n
0 P | 1 y | 2 t | 3 h | 4 o | 5 n |
```

Loops control statements in Python are:

break: Terminates the loop statement and transfers execution to the statement immediately following the loop.

continue: Causes the loop to skip the remainder of its body and immediately retest it’s condition prior to reiterating.

pass: Used when a statement is required syntactically but you do not want any command or code to execute.

Example

```python
for letter in 'Python':
    if letter == 'o':
        break
    if letter == 't':
        continue
    if letter == 'P':
        pass
    else:
        print(letter)
```

Output

```
y
h
```

Functions

Python **functions** syntax is like:

```
def person_data(name, age, current_year=2019):
    """function doc string, this function return year of birth"""
    born_on = current_year - age
    return "{} born on {}".format(name, born_on)
```

In this function **name** and **age** are **Required-Arguments** and **current_year** is **Optional-Argument** with **default** value.
this function can be called like:

```
person_data("Abolfazl", 24, 2020) # name, age and current_year are given
person_data("Abolfazl", 23) # current_year keep it’s default value 2019
person_data(age=23, name="Abolfazl") # name and age are given as keywords
```

Also arguments can be sent like **tuples** or **dictionaries** too:

```
def print_arguments(arg, *args, **kwargs):
    print("arg = ", arg)
    print("args = ", args)
    print("kwargs = ", kwargs)
print_arguments(1, 2, 3, 4, name="Abolfazl", family="Amiri")
```

The output would be:

```
arg =  1
args =  (2, 3, 4)
kwargs =  {'name': 'Abolfazl', 'family': 'Amiri'}
```

Working With Files

Opening a file, writing some data and then printing it’s content:

```python
# use 'w' for write mode and 'a' for append mode.
# these modes will create file if not exist
file_object = open(file='example.txt', mode='a')
file_object = open(file='example.txt', mode='w')

# writing to file
file_object.write('write this lines!\nto the file.\n')
file_object.writelines(['line 3\n', 'line 4\n', 'line 6'])

# files most be closed to save changes
file_object.close()

# use 'r' for read mode.
# this mode will raise FileNotFoundError Exception if file not exist
file_object = open(file='example.txt', mode='r')

# reading file content
print("readline  output >>", file_object.readline(), end='')
print("readlines output >>", file_object.readlines())
print("read      output >>", file_object.read(), end='')
file_object.close()
```

Output:

```
readline  output >> write this lines!
readlines output >> ['to the file.\n', 'line 3\n', 'line 4\n', 'line 6']
read      output >>
```

Exception Handling

In Python all **exceptions** are sub-classes of **Exception** class.

Trying to open a file that does not exist:

```
try:
    file = open('example2.txt', 'r')
    print('file content >>', file.read())
    file.close()
except NameError:
    print('NameError raised!')
except Exception as e:
    print(e)
finally:
    print('this will be printed anyway!')
```

Output:

```
[Errno 2] No such file or directory: 'example2.txt'
this will be printed anyway!
```

Modules

A **module** is a **file** consisting of Python code that can define **functions**, **classes** and **variables**.

You can use **any Python source file as a module** by executing an import statement.

```
import datetime
```

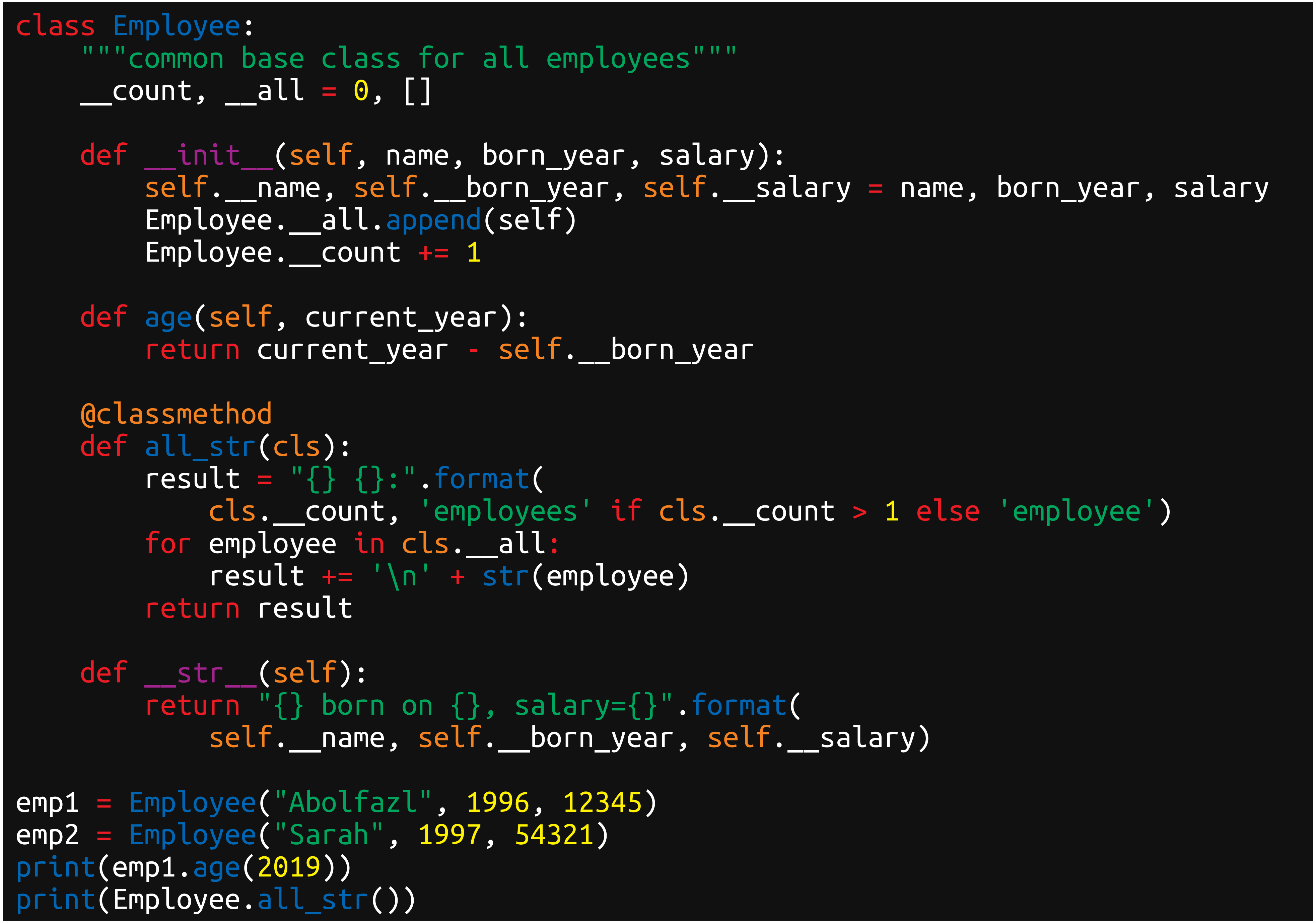
Python's **from** statement lets you import **specific attributes** from a module into the current **namespace**.

```
from datetime import datetime, timezone
from json.encoder import JSONEncoder
```

import * statement can be used to import **all names** from a module into the current **namespace**.

```
from datetime import *
```

Object Oriented Programming



```python
class Employee:
    """common base class for all employees"""
    __count, __all = 0, []

    def __init__(self, name, born_year, salary):
        self.__name, self.__born_year, self.__salary = name, born_year, salary
        Employee.__all.append(self)
        Employee.__count += 1

    def age(self, current_year):
        return current_year - self.__born_year

    @classmethod
    def all_str(cls):
        result = "{} {}:".format(
            cls.__count, 'employees' if cls.__count > 1 else 'employee')
        for employee in cls.__all:
            result += '\n' + str(employee)
        return result

    def __str__(self):
        return "{} born on {}, salary={}".format(
            self.__name, self.__born_year, self.__salary)

emp1 = Employee("Abolfazl", 1996, 12345)
emp2 = Employee("Sarah", 1997, 54321)
print(emp1.age(2019))
print(Employee.all_str())
```

Output:

```
23
2 employees:
Abolfazl born on 1996, salary=12345
Sarah born on 1997, salary=54321
```

Built-in class functions:

```python
# return 'True' if emp1 has 'age' attribute otherwise 'False'
hasattr(emp1, 'age')
# return 'age' attribute value
getattr(emp1, 'age')
# set 'age' attribute value to 24
setattr(emp1, 'age', 24)
# delete 'age' attribute from emp1 object
delattr(emp1, 'age')
```

method/attribute started with **double-underscore (__)** is **private** to the class and will not be inherited from subclass and is accessible with **_class__attribute** inside subclass.

method/attribute started with **underscore (_)** is **protected**, but is accessible from subclass and directly.

Inheritance:

```python
class Person:
    def __init__(self, name, age):
        self.name, self.__age = name, age

    def get_details(self):
        return "name={} age={}".format(self.name, self.__age)

class Student(Person):
    def __init__(self, name, age, branch, year):
        self.branch, self.year = branch, year
        # also 'Person.__inti__(name, age)' can be used
        super().__init__(name, age)

    def get_details(self):
        return "name={} age={} branch={} year={}".format(
            self.name, self._Person__age, self.branch, self.year)

person = Person('Sarah', 23)
student = Student('Abolfazl', 24, 'CSE', 2014)
print(person.get_details())
print(student.get_details())
```

Output:

```
name=Sarah age=23
name=Abolfazl age=24 branch=CSE year=2014
```

Tips & Tools

Python scripts can be written with any **text-editors** such **vim** or **notepad**, also a Python Development Plugin is almost available for any IDE, such **PyDev for Eclipse**.

PyCharm is a Python IDE with Community and Professional editions.

```
https://www.jetbrains.com/pycharm/
```

Some useful Python **standard libraries**:

- **os** >> operating system interfaces
- **datetime** >> basic date and time types
- **math** >> mathematical functions
- **random** >> generate pseudo-random numbers
- **sqlite3** >> interface for SQLite databases
- **hashlib** >> secure hashes and message digests
- **threading** >> thread-based parallelism
- **subprocess** >> subprocess management
- **tkinter** >> python interface to Tcl/Tk
- **unittest** >> unit testing framework

find more here:

```
https://docs.python.org/3/library/
```

References

You can find this **file** and **example** script files in:

```
https://abolfazlamiri.ir/python3tutorials/
https://github.com/aasmpro/python3tutorials/
```

Python official **documents**:

```
https://docs.python.org/3/
```