GEEKINSTA

[GEEKINSTA]
[Python 3 cheat sheet for beginners]
[https://www.geekinsta.com]

Basics

The print() function displays the given message on the screen.

```
print("Hello World")
# Hello World
print(20+5)
# 25
print(f'10 + 5 is {10+5}')
# 10 + 5 is 15
print("python" * 2)
# python python
a, b, c = 1, 2, 3
```

Variables

In programming, a variable is a value that can be changed. All variables will have a storage location and a name. The variable name is usually used to refer to the value that is stored in a variable.

In Python, a variable is declared when you set a value to the variable. Unlike some other programming languages, Python does not have special keywords for declaring a variable.

Read more about Python variables.

```
a = 10  # int
b = 10.5  # float
c = "Python"  # string
d = 'p'  # string
e = 10j  # complex
f = True  # bool
```

Type conversion

We perform type conversions to convert a variable from one data type to another.

Generally, there are two types of type conversions.

- **Implicit conversions** – This conversion is type-safe and requires no special code. It is performed by the compiler and no data loss occurs.
- **Explicit conversions** – These conversions require a cast operator. Explicit conversion cannot be made without the risk of losing information. A cast is a way of informing the compiler that you wish to make a conversion and you are aware of the possible data loss.

```
i = int("5")  # 5
i = float(1)  # 1.0
i = bool(10)  # True
i = bool(0)  # False
i = str(10)  # 10
```

Conditional statements

These statements are used to make decisions based on one or more conditions. If the condition specified in a conditional statement is found true, then a set of instructions will be executed and otherwise, something else will be executed.

```
if a == 0:
    print('a = 0')
elif a == 1:
    print('a = 1')
else:
    print('a')
a, b, c = 1, 2, 3
# if a>b>c:
if a>b and a>c:
    print('A is largest')
elif b>c:
    print('B is largest')
else:
    print('C is largest')
```

Ternary operator

It is a special conditional operator that compares two values and determines the third value based on the comparison.

```
a = 20
x = True if a >= 1 else False
# x = False
print('Yes') if 11 > 10 else print("No")  # Yes
a, b = 10, 11
r = 'Yes' if a >= b else 'No'
print(r)  # No
```

Loops

With the help of a loop or looping statements, you can execute a statement or a set of statements until an expression is evaluated to false.

```
# For loop
# for i in range(5)
for i in range(0, 5):
    print(i)

for i in range(4, -1, -1):
    print(i)

# While loop
i=0
while i <= 2:
    print(i)
    i+=1
```

Functions

A function contains a set of statements that creates output based on the given input or parameters.

```
# Empty function
def myfun():
    pass

def myfun1():
    print("hi")

myfun()  # No output
myfun1()  # hi
def add(a, b):
    return a + b

add(1, 2)  # 3

def calc(a, b):
    return a+b, a-b
```

```
res = calc(5, 2)
print("After addition", res[0])  # After addition 7
print("After substraction", res[1])  # After substraction 3
def msg(name="John Doe"):
    print("Hi", name)

msg()  # Hi John Doe
msg("Jane")  # Hi Jane

def fnvar(*data):
    print(data[0])

fnvar(1, 2, 3)  # 1

def fnkwargs(**data):
    print(data['name'])

fnkwargs(name='John Doe', age=20)  # John Doe
```

List

In Python, a list is an ordered collection of items of different data types. The items of a list are enclosed in [ ] brackets. Each list item is separated by a comma (,).

Create a list

```
# Creating a list
lst = [0, 1, 2]
print(lst[0])  # 0
print(lst[0:2])  # [0, 1]
lst = ['1'] * 3
print(lst)  # ['1', '1', '1']
```

Add elements

```
lst = [1, 2, 3]
lst.append('last')  # [1, 2, 3, 'last']
lst.insert(0, 'first')  # ['first', 1, 2, 3, 'last']
lst.insert(20, 'New element')  # ['first', 1, 2, 3, 'last', 'New element']
```

Remove elements

```
lst = [1, 2, 3, 4, 5, 2, 8, 9]
print(lst.pop())  # Removes and returns the last element
print(lst.pop(0))  # Removes and returns element at index 0
lst.remove(2)  # Removes the first occurrence of 2 (Remove by value)

del lst[0]  # Removes element at index 0
del lst  # Deletes the list
```

Modify elements

```
lst = [1, 2, 3]
lst[0] = 5  # [5, 2, 3]
```

Sorting

```
# Sort
lst = [0, 5, 3, 2, 4, 1]
lst.sort()  # [0, 1, 2, 3, 4, 5]
lst.sort(reverse=True)  # [5, 4, 3, 2, 1, 0]

# Custom sorting
ages = [
    ["John", 20],
    ["Jane", 22],
    ["Janet", 21]
]
# Sorts based on age
ages.sort(key=lambda age: age[1])
```

```
print(ages)  # [['John', 20], ['Janet', 21], ['Jane', 22]]
```

Tuple

A list is an ordered collection of items of various data types. The items of a tuple are enclosed in () brackets. Each list item is separated by a comma (,). Unlike a list, the elements of a tuple cannot be modified.

```
tpl = (1, 2, 3)
print(tpl[0])  # 1
print(len(tpl))  # 3
print(tpl[0:2])  # (1, 2)
```

Dictionary

A dictionary is a collection of data as key-value pairs. Each item of a dictionary is contained in {} brackets.

Create a dictionary

```
d = {"name": 'python', "version": 3.7, 0: 1}
print(d['name'])  # python
print(d[0])  # 1
print(len(d))  # 3
```

Add elements

```
d = {"name": 'python'}
d['version'] = '3.7'  # Add element
d.update({"type": "dynamic", "oop": "yes"})  # Add multiple elements
```

Remove elements

```
d = {"name": "python", "type": "dynamic", "year": 1991}
print(d.pop("year"))  # Removes and returns the item
del d["type"]  # Removes the item
print(d)  # {'name': 'python'}
d.clear()  # Removes all the elements
```

Sets

Similar to a dictionary, the elements of a set are contained in {} brackets. The main difference between sets and other datatypes like list, tuple, and dictionary is that sets do not have an index.

Create a set

```
s = {5, 2, 1, 4}
print(s)  # {1, 2, 4, 5}
```

Add / remove items

```
s = {5, 2, 1, 4}
s.add(55)
print(s)  # {1, 2, 4, 5, 55}
print(s.pop())  # 1
```

Comprehension

Comprehensions in Python provide us with a short and succinct way to create new lists, set, or dictionary using iterable objects that have been already defined. Python supports the following comprehensions:

- List Comprehensions
- Dictionary Comprehensions

- Set Comprehensions

```
# List comprehension
lst = [x for x in range(1, 6)]
# [1, 2, 3, 4, 5]

# Set comprehension
st = {x for x in range(1, 6)}
# {1, 2, 3, 4, 5}

# Dictionary comprehension
dict = {x: x*2 for x in range(1, 6)}
# {1: 2, 2: 4, 3: 6, 4: 8, 5: 10}
```

Generator expression

Similar to comprehensions, generator expressions are also a convenient way to create iterable objects.

```
res = (x for x in range(6))
print(res)  # <generator object <genexpr> at 0x000001E65CD5BE08>
print(res[0])  # Error
print(len(res))  # Error
lst = list(res)  # Converts to list
```

Unpacking

Unpacking is a method by which we extract the values of an iterable object to separate variables.

```
a = [1, 2, 3]
b = [4, 5, 6]
c = [*a, *b, "unpacked"]
```

```
print(c)  # [1, 2, 3, 4, 5, 6, 'unpacked']

x = {"x": 1}
y = {"y": 2}
z = {**x, **y}
print(z)  # {'x': 1, 'y': 2}
a = [1, 2, 3]
def myfun(a, b, c):
    print(a+b+c)

# myfun(a)  # Error
myfun(*a)
```

Lambda

A **lambda** expression is a special syntax to create functions without names.

```
x = lambda a, b, c: a + b + c
print(x(5, 6, 2))  # 13
def myfunc(b):
    return lambda a: a * b

dbl = myfunc(2)
print(dbl(11))  # => 22
```

Datetime module

Python DateTime module comes with everything you need to access date and time.

```
import datetime
a=datetime.date.today()
print(a.strftime("%B/%A/%Y/%D"))  # August/Friday/2019/08/23/19
```

Exception handling

Exception handling is the mechanism by which errors in the application are captured and handled. It is one of the most important features of a programming language.

In Python, Exception handling is done with the help of the following keywords:

- try
- except
- finally
- raise

```
try:
    a = 1/0
except:
    print('Division by zero is not possible')
try:
    print(a)
except NameError:
    print('Variable is not declared')
try:
    10/0
except ArithmeticError:
    print('An arithmetic error has occurred')
finally:
    print('I will be executed even if there is no error')
try:
    raise Exception("An error occurred")
finally:
    print("Error occured")

try:
    raise ArithmeticError("An error has occurred")
except Exception as e:
```

```
    print(e)
try:
    10/0
except Exception as e:
    print(type(e).__name__)
try:
    a = int(input("Enter a positive number"))
    if a < 0:
        raise ValueError("Number is not valid")
    if a == 0:
        raise ArithmeticError("Division by zero error")
except (ValueError , ArithmeticError):
    print("A value error or arithmetic error has occured")
```

Class

A class is a blueprint of an object. An object in the real world will have properties like shape, color, weight, etc. Likewise, in object-oriented programming, a class defines certain properties, events, and functionalities that an object can have.

```
# Class
class MyClass:
    name = "Hi from the class"

    # Constructor
    def __init__(self):
        self.age = 22
        gender = 'Male'

    # Method
    def show_data(self):
        print(self.name)
        print(self.age)
        # print(gender)  # Error
```

```
# Creating object
obj = MyClass()
obj.show_data()
```

Single inheritance

Single inheritance enables a derived class to inherit properties and behavior from a single parent class.

```
class A:

    def fun1(self):
        print('Function 1')

class B (A):

    def fun2(self):
        print('Function 2')

obj = B()
obj.fun1()
obj.fun2()
```

Multiple inheritance

Single inheritance enables a derived class to inherit properties and behavior from multiple classes.

```
class A:

    __m = 'Private'
    def __init__(self):
        self.msg = 'Class 1'

    def fun1(self):
        print('Function 1')
```

```
    def same(self):
        print('Same A')

class B:

    def fun2(self):
        print('Function 2')

    def same(self):
        print('Same B')

class C (A, B):

    def fun3(self):
        self.fun1()
        self.fun2()
        self.same()
        # print(self.__m)  # Cannot access private members

    # def same(self):  # If uncommented, this method will be executed
    #    print("override")

obj = C()
obj.fun3()
```

Multilevel inheritance

Multiple inheritance means that a class is inheriting the properties of a derived class.

```
class A:

    __m = 'Private'
    def __init__(self):
```

```
        self.msg = 'Class 1'

    def fun1(self):
        print('Function 1')

class B (A):

    def fun2(self):
        print('Function 2')

class C (B):

    def fun3(self):
        self.fun1()
        self.fun2()
        # print(self.__m)

obj = C()
obj.fun3()
```

Abstract classes

Generally, an abstract class is a class that contains abstract methods and cannot be instantiated. It serves as a blueprint for other classes expecting its child classes to implement its abstract methods.

Python by default does not provide **abstract** classes. But **it** has a module named ABC which provides the base for defining **Abstract** Base classes(ABC).

```
from abc import ABC, abstractmethod

class Class1 (ABC):

    @abstractmethod
    def showmessage(self):
        print('From base class')
```

```
    @abstractmethod
    def logic(self):
        print('Logic in the derived class')

    @abstractmethod
    def errorchecking(self):
        print('You should implement logic in the child class')

class Class2(Class1):

    def showmessage(self):
        super().showmessage()

    def logic(self):
        print('Custom logic')

    def errorchecking(self):
        print('Try commenting this method')

obj = Class2()
obj.showmessage()
obj.logic()
obj.errorchecking()
```

Attribute methods

These methods allow you to get or check the properties of an object.

```
class MyClass:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def showdata(self):
        print(f'Name: {self.name}, Age: {self.age}')
```

```
cls = MyClass('John Doe', 25)
print(getattr(cls, 'name'))
setattr(cls, 'age', 22)
print(getattr(cls, 'age'))
print(hasattr(cls, 'name'))
delattr(cls, 'age')
print(hasattr(cls, 'age'))
```

Class methods

Class methods are not specific to any instances. So, they can be accessed without creating an object. It can access and modify the state of a class.

```
class MyClass:
    @classmethod
    def fun1(self):
        print('Hello World')

MyClass.fun1()  # THE METHOD CAN BE CALLED WITHOUT CREATING AN OBJECT
```

Class methods can access and modify the state of an instance.

```
class MyClass:
    a = 20
    @classmethod
    def fun1(self):
        self.a = 21

    def show_data(self):
        print(self.a)

obj = MyClass()
obj.show_data()  # 20
MyClass.fun1()
```

```
obj.show_data()  # 21
```

Static methods

Like class methods, static methods can also be called without creating an object. But it cannot access or modify the state of a class. Also, it does not accept the class parameter and is created only once.

```
class MyClass:

    a = 'Static method will print this'
    @staticmethod
    def my_static_method():
        print('Hi from static method')
        print(MyClass.a)

    def my_method(self):
        self.a = 'Instance method'
        print(f'Hi from {self.a}')

cls = MyClass()
cls.my_method()
MyClass.my_static_method()
# Hi from Instance method
# Hi from static method
# Static method will print this
```

Private methods

Private methods are methods that cannot be accessed outside the class and by child classes using inheritance.

```
class SeeMee:

    def youcanseeme(self):
        return 'you can see me'
```

```
    def __youcannotseeme(self):
        return 'you cannot see me'


Check = SeeMee()
print(Check.youcanseeme())
# you can see me
# print(Check.__youcannotseeme())
# AttributeError: 'SeeMee' object has no attribute '__youcannotseeme'


##########BUT YOU CAN STILL ACCESS IT################
print(Check._SeeMee__youcannotseeme())
```