

LEARNING PROGRESS REVIEW

Data Engineer Batch 9 - Week #2

PYTHON PROGRAMMING AND GIT & GITHUB

Dwi Handoyo

PRIMITIVE DATA TYPES

- * **Integer** (int), are all integers: 1, -2304, 0, etc.
- * **Float** (float), are all decimal numbers: 23.5, -0.52, 3525.252, etc.
- * **String** (str), are all letters and text: "a", "This is text. And another sentence.", etc.
- * **Boolean** (bool), is True or False.

Check data type in google colab or Python IDE:

```
a = 2.51  
print(type(a))
```

<class 'float'>

```
b = "2.51"  
print(type(b))
```

<class 'str'>

```
c = True  
print(type(c))
```

<class 'bool'>

```
d = "True"  
print(type(d))
```

<class 'str'>

NON PRIMITIVE DATA TYPES

- * **List** - Characteristics: mutable data (changeable), ordered, and allow duplicates

```
list = [3,5,2,7]
```

- * **Tuple** - Characteristics: immutable, ordered, and allow duplicates

```
tuple = ('p','e','r','m','i','t')
```

- * **Set** - Characteristics: unchangeable (but items can be added/deleted), unordered/unindexed, and unique.

```
set = {1, 2, 3}
```

- * **Dictionary** - Characteristics: mutable, ordered, keys have to be unique

```
dict = {1: 'apple', 2: 'ball', 3: 'car'}
```

Python also has a module called datetime for working with dates and times data type

Check data type in google colab or Python IDE:

```
a = [3,5,2,7]
```

```
print(type(a))
```

```
<class 'list'>
```

```
b = ('p','e','r','m','i','t')
```

```
print(type(b))
```

```
<class 'tuple'>
```

```
c = {1, 2, 3}
```

```
print(type(c))
```

```
<class 'set'>
```

```
d = {1: 'apple', 2: 'ball', 3: 'car'}
```

```
print(type(d))
```

```
<class 'dict'>
```

COMMENT & DOCSTRING

Comments

Use # prior comments

Single Line Comment

```
#simple math operation
```

```
x = 2 + 3
```

```
print(x)
```

Multiline Comment

```
# prints text block to describe
```

```
# your favorite pet
```

```
named = "garfield"
```

```
species = "cat"
```

```
breed = "british shorthair"
```

```
color = "chocolate"
```

Docstring

Use triple quotes for docstring

```
def square(n):
```

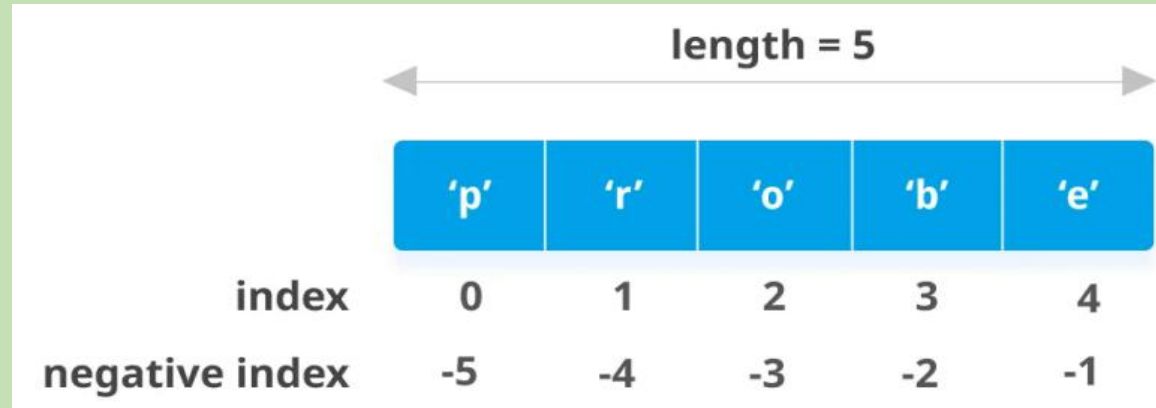
```
    """Takes in a number n, returns the square of n"""
```

```
    return n**2
```

```
square.__doc__
```

```
    """Takes in a number n, returns the square of n"""
```

INDEXING



SUBSETTING

String

```
str = 'September'  
x = str[-1]  
print(x)
```

r

List

```
list = ['one', 'two', 'three', 'four', 'five']  
x = list[2]  
print(x)
```

three

Tuple

```
tuple = (4, 17, 6, 5, 9, 10)  
x = tuple[0]  
print(x)
```

4

SLICING

String

```
str = 'September'  
x = str[:4]  
print(x)
```

Sept

List

```
list = ['one', 'two', 'three', 'four', 'five']  
x = list[0:3]  
print(x)
```

['one', 'two', 'three']

Tuple

```
tuple = (4, 17, 6, 5, 9, 10)  
x = tuple[3:]  
print(x)
```

(5, 9, 10)

ASSIGNMENT

List

```
list = [3,4,6,9,0,3,2]  
list[1:4] = ['a','b','c']  
print(list)
```

[3, 'a', 'b', 'c', 0, 3, 2]

N ested List

```
list = [3,4,6,9,0,3,2]  
list[4] = ['a','b','c']  
print(list)
```

[3, 4, 6, 9, ['a', 'b', 'c'], 3, 2]

Tuple Re-assigned

```
tuple = (5, 3, 9, 1)  
print(tuple)  
tuple = ('a','b','c')  
print(tuple)
```

(5, 3, 9, 1)
('a', 'b', 'c')

List Nested Tuple

```
tuple = (5, 3, 9, ['a', 'b'], 1)  
tuple[3][1] = 'c'  
print(tuple)
```

(5, 3, 9, ['a', 'c'], 1)

ASSIGNMENT

N ested Dictionary

```
fruit = {'apple': {'red' : 3, 'yellow' : 0, 'green' : 2}, 'orange':  
{ 'red' : 0, 'yellow' : 4, 'green' : 1}}  
fruit['orange']['yellow'] = 7  
print(fruit)
```

```
{'apple': {'red': 3, 'yellow': 0, 'green': 2}, 'orange': {'red': 0,  
'yellow': 7, 'green': 1}}
```

change Dictionary value

```
my_dict = {'name': 'Jack',  
'age': 26}  
my_dict['age'] = 27  
print(my_dict)
```

```
{'name': 'Jack', 'age': 27}
```

add Dictionary key & value

```
my_dict = {'name': 'Jack', 'age':  
26}  
my_dict['address'] = 'Downtown'  
print(my_dict)
```

```
{'name': 'Jack', 'age': 26,  
'address': 'Downtown'}
```

update Dictionary 1

```
fruit = {'apple': 2, 'orange': 7, 'manggo': 9}  
fruit.update({'orange': 3})  
print(fruit)
```

```
{'apple': 2, 'orange': 3, 'manggo': 9}
```

update Dictionary 2

```
fruit = {'apple': 2, 'manggo': 9}  
fruit.update({'pineapple': 7})  
print(fruit)
```

```
{'apple': 2, 'manggo': 9, 'pineapple': 7}
```

update Dictionary 3

```
fruit = {'apple': 2, 'manggo': 9}  
fruit.update([('lychee', 6)])  
print(fruit)
```

```
{'apple': 2, 'manggo': 9, 'lychee': 6}
```

LENGTH

String

```
str = 'September'  
x = len(str)  
print(x)
```

9

List

```
list = ['one', 'two', 'three',  
        'four', 'five']  
x = len(list)  
print(x)
```

5

Tuple

```
tuple = (4, 17, 6, 5, 9,  
         10)  
x = len(tuple)  
print(x)
```

6

Set

```
set = {1,2,3,4}  
x = len(set)  
print(x)
```

4

Dictionary

```
dic = {'apple': 2, 'orange': 7,  
       'manggo': 9}  
x = len(dic)  
print(x)
```

3

REMOVING ELEMENT

remove from List

```
list_1 = [3,4,6,9]  
list_1.remove(4)  
print(list_1)
```

[3, 6, 9]

delete items in List

```
list_1 = [3,4,6,9,'a','b']  
del list_1[3:5]  
print(list_1)
```

[3, 4, 6, 'b']

pop in List

```
list_1 = [3,4,6,9]  
list_1.pop(2)  
print(list_1)
```

[3, 4, 9]

clear List

```
list_1 = [3,4,6,9]  
list_1.clear()  
print(list_1)
```

[]

assign blank List

```
list_1 = [3,4,6,9]  
list_1[1:3] = []  
print(list_1)
```

[3, 9]

REMOVING ELEMENT

delete List

```
lst = [4,6,3,7]
del lst
print(lst)
```

NameError: name 'lst' is not defined

delete Tuple

```
tpl = (4,6,3,7)
del tpl
print(tpl)
```

NameError: name 'tpl' is not defined

delete Dictionary

```
dic = {'apple': 2, 'orange': 7, 'manggo': 9}
del dic
print(dic)
```

NameError: name 'dic' is not defined

pop in Dictionary

```
dict = {'apple': 2, 'orange': 7,
'manggo': 9}
dict.pop('orange')
print(dict)
```

```
{'apple': 2, 'manggo': 9}
```

popitem in Dictionary

```
dict = {'apple': 2, 'orange': 7,
'manggo': 9}
print(dict.popitem())
print(dict)
```

```
('manggo', 9)
{'apple': 2, 'orange': 7}
```

clear Dictionary

```
dict = {'apple': 2, 'orange': 7,
'manggo': 9}
dict.clear()
print(dict)
```

```
{ }
```

REMOVING ELEMENT

discard Set

```
my_set = {7, 8, 9, 10, 11}
```

```
my_set.discard(10)  
print(my_set)
```

```
{7, 8, 9, 11}
```

pop Set

```
my_set = {7, 8, 9, 10, 11}
```

```
print(my_set.pop())  
print(my_set)
```

```
7  
{8, 9, 10, 11}
```

remove Set

```
my_set = {7, 8, 9, 10, 11}
```

```
my_set.remove(8)  
print(my_set)
```

```
{7, 9, 10, 11}
```

clear Set

```
my_set = {7, 8, 9, 10, 11}
```

```
my_set.clear()  
print(my_set)
```

```
set()
```

ADDITION

List Concatenation

```
list_1 = [3,4,6,9]
list = list_1 + ['a','b','c']
print(list)
```

[3, 4, 6, 9, 'a', 'b', 'c']

List Repetition

```
list_1 = [3,4,6,9]
list = list_1 * 2
print(list)
```

[3, 4, 6, 9, 3, 4, 6, 9]

List insert()

```
list = [1, 9, 4, 5]
list.insert(2,3)
print(list)
```

[1, 9, 3, 4, 5]

List append()

```
list = [1, 9, 8]
list.append(3)
print(list)
```

[1, 9, 8, 3]

String Concatenation

```
str1 = 'Hello'
str2 = ' World!'
str = str1 + str2
print(str)
```

Hello World!

String Repetition

```
str1 = 'Hello'
str2 = ' World!'
str = str1 * 3
```

HelloHelloHello

Tuple Concatenation

```
tuple1 = (5, 3, 9, 1)
tuple2 = ('a','b','c')
tuple = tuple1 + tuple2
print(tuple)
```

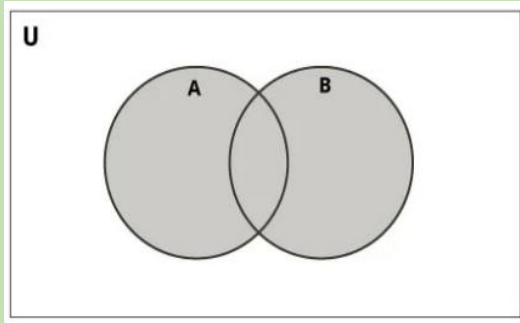
(5, 3, 9, 1, 'a', 'b', 'c')

Tuple Repetition

```
tuple1 = (5, 3, 9, 1)
tuple2 = ('a','b','c')
tuple = tuple2 * 2
print(tuple)
```

('a', 'b', 'c', 'a', 'b', 'c')

ADDITION

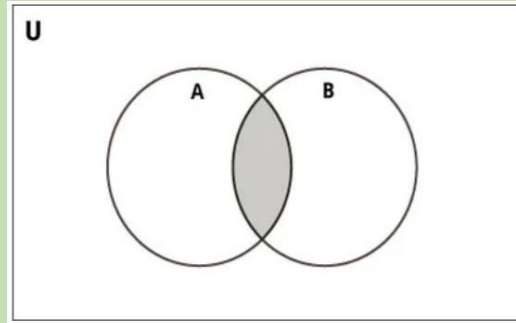


Set Union

$A = \{1, 2, 3, 4, 5\}$
 $B = \{4, 5, 6, 7, 8\}$

```
print(A | B)
print(A.union(B))
print(B.union(A))
```

$\{1, 2, 3, 4, 5, 6, 7, 8\}$
 $\{1, 2, 3, 4, 5, 6, 7, 8\}$
 $\{1, 2, 3, 4, 5, 6, 7, 8\}$

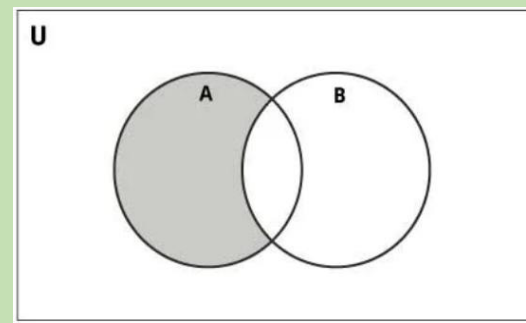


Set Intersection

$A = \{1, 2, 3, 4, 5\}$
 $B = \{4, 5, 6, 7, 8\}$

```
print(A & B)
print(A.intersection(B))
print(B.intersection(A))
```

$\{4, 5\}$
 $\{4, 5\}$
 $\{4, 5\}$

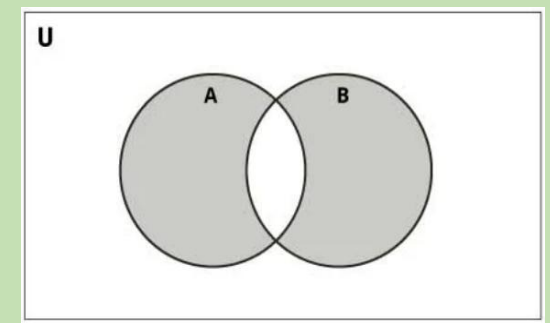


Set Difference

$A = \{1, 2, 3, 4, 5\}$
 $B = \{4, 5, 6, 7, 8\}$

```
print(A - B)
print(A.difference(B))
print(B - A)
print(B.difference(A))
```

$\{1, 2, 3\}$
 $\{1, 2, 3\}$
 $\{8, 6, 7\}$
 $\{8, 6, 7\}$



Set Symetric Difference

$A = \{1, 2, 3, 4, 5\}$
 $B = \{4, 5, 6, 7, 8\}$

```
print(A ^ B)
print(A.symmetric_difference(B))
print(B.symmetric_difference(A))
```

$\{1, 2, 3, 6, 7, 8\}$
 $\{1, 2, 3, 6, 7, 8\}$
 $\{1, 2, 3, 6, 7, 8\}$

PYTHON OPERATORS

Comparison Operators

Operators	Descriptions	Example
>	Strictly Less than	$x < 9$ is True if x is less than 9
<	Strictly Greater than	$x > y$ is True if x is greater than y
==	Equal	$x == 13$ is True if and only if x is 13
!=	Not equal	$x != y$ is True if and only if x is not y
<=	Less than or equal	$x \leq 23$ is True if x is 23 or less
>=	Greater than or equal	$x \geq y$ is True if x is y or greater

Boolean Operators: and

A	B	A and B
True	True	True
True	False	False
True	False	False
False	False	False

Boolean Operators: or

A	B	A or B
True	True	True
True	False	True
True	False	True
False	False	False

Boolean Operators: not

A	not A
True	False
False	True

PYTHON CONDITIONS

Single Condition (If)

Example:

```
a = 3  
if a == 3:  
    print("a is three")
```

output: a is three

Two Condition (If -> Else)

Example:

```
a = 5  
b = 5  
if a != b:  
    print("a and b is not the same")  
else:  
    print("a and b is the same")
```

output: a and b is the same

Multiple Condition (If -> Elif -> Else)

Example:

```
a = 3  
b = 5  
if a > b:  
    print("a is bigger than b")  
elif a < b:  
    print("a is smaller than b")  
else:  
    print("a and b is the same")
```

output: a is smaller than b

PYTHON CONDITIONS

Single Expression

Example:

```
a = 5
b = 7
if a <= b:
    print("a is smaller than b")
```

output: a is smaller than b

Multiple Expression

Example 1 (and):

```
a = 3
b = 7
c = 20
if a < b and b < c:
    print("a is smaller than b")
```

output: a is smaller than b

Example 2 (or):

```
a = 3
b = 7
c = 20
if c > b or c < a:
    print("At least one of the expression is True")
```

output: At least one of the expression is True

PYTHON CONDITIONS

If Statement Shorthand

If Statement	Shorthand
If a = 3 if a == 3: print("a is three")	a = 3 if a == 3: print("a is three")
If Else a = 5 b = 5 if a == b: print("same") else: print("different")	a = 5 b = 5 print("same") if a==b else print("different")

Nested If

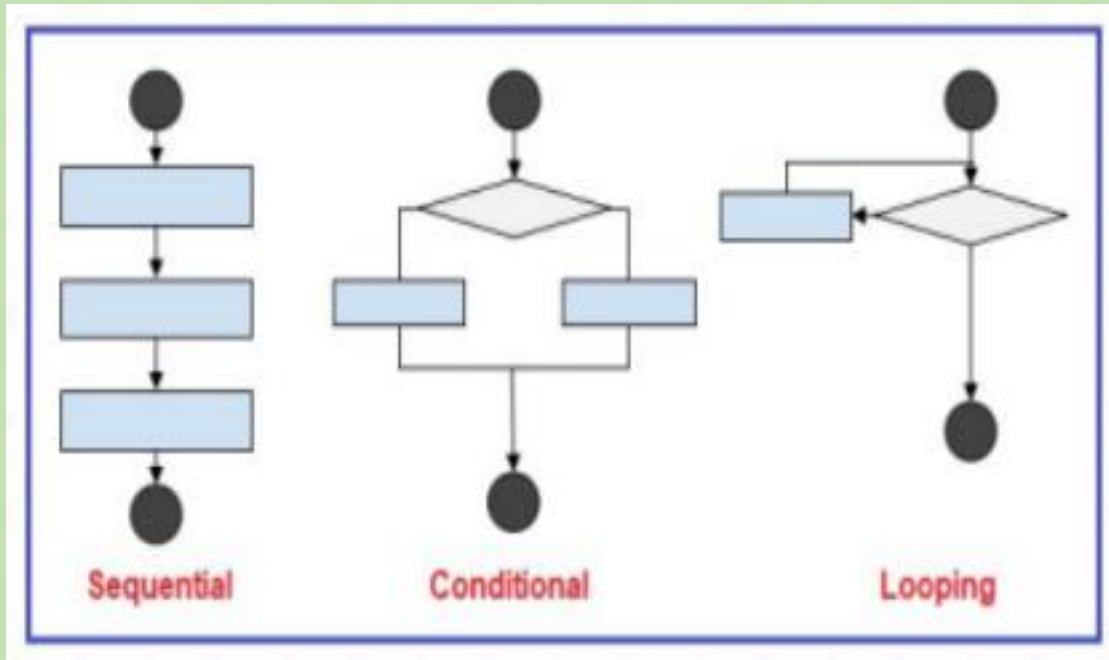
Example:

```
x = 50
if x > 20:
    print("Above twenty, ")
    if x > 40:
        print("and also above 40")
    else:
        print("but not above 40")
else:
    print("Below twenty")
```

Output: Above twenty, and also above 40

CONTROL FLOW : SEQUENTIAL & CONDITIONAL

Flow Chart Types



Sequential Flow

Example:

```
a=20  
b=10  
c=a-b  
print("a-b =",c)
```

Output: a-b = 10

Conditional Flow

Example:

```
a = 3  
b = 5  
if a > b:  
    print("a is bigger than b")  
else:  
    print("a is smaller than b")  
    print("Done")
```

Output:

*a is smaller than b
Done*

CONTROL FLOW : LOOPING (WHILE)

While Loop with Else

Example:

```
count = 0
while count < 3:
    print(count, " is less than 3")
    count = count + 1
else:
    print(count, " is not less than 3")
```

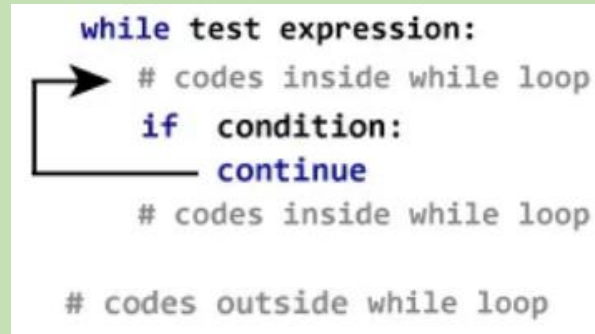
0 is less than 3
1 is less than 3
2 is less than 3
3 is not less than 3

While Loop with Continue

Example:

```
x = 3
while x < 10:
    x += 1
    if x > 7:
        continue
    print(x)
```

4
5
6
7

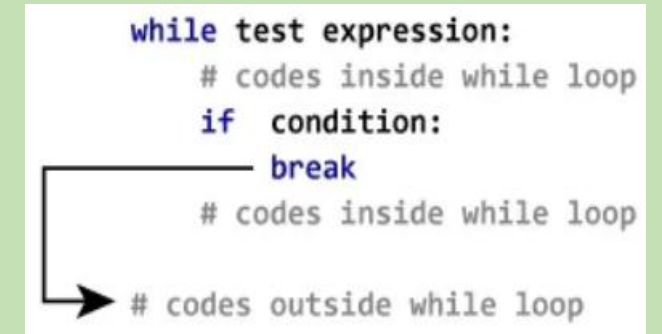


While Loop with Break

Example:

```
i = 1
while i <= 100 :
    print(i)
    if i == 7 :
        break
    i += 1
```

1
2
3
4
5
6
7



CONTROL FLOW : LOOPING (FOR)

String For Loop

Example:

```
if __name__ == "__main__":  
    s = "Hello"  
    for c in s:  
        print(c)
```

H
e
l
l
o

Enumerate

Example:

```
str = 'python'
```

```
list_enumerate = list(enumerate(str))  
print(list_enumerate)  
[(0, 'p'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
```

List, Set, & Tuple For Loop

Example:

```
if __name__ == "__main__":  
    l = ['a', 'b', 'c']  
    s = {'a', 'b', 'c'}  
    t = ('a', 'b', 'c')  
    print("list:")  
    for c in l:  
        print(c)  
    print("set:")  
    for c in s:  
        print(c)  
    print("tuple:")  
    for c in t:  
        print(c)
```

list:	set:	tuple:
a	a	a
b	c	b
c	b	c

For Loop on Data Structures (List, Tuple, Set, Dictionary)

Dictionary For Loop

Example:

```
if __name__ == "__main__":  
    d = {'a': 0, 'b': 1, 'c': 2, 'd': 3}  
    print("keys:")  
    for k in d.keys():  
        print(k)  
    print("values:")  
    for v in d.values():  
        print(v)  
    print("key-value pairs:")  
    for k, v in d.items():  
        print(k, "->", v)
```

keys:
a
b
c
d
values:
0
1
2
3
key-value pairs:
a -> 0
b -> 1
c -> 2
d -> 3

CONTROL FLOW : LOOPING (FOR)

Range()

The range() object is "lazy" because it will not output any numbers in it when we create it. This function does not store all values in memory. So it only remembers start, stop, step size and generate the next number on the go. To force this function to display all items, we can use function list()

```
print(range(10))  
print(list(range(10)))  
print(list(range(2, 8)))  
print(list(range(2, 20, 3)))
```

Iterate through a list using indexing

```
genre = ['pop', 'rock', 'jazz']
```

iterate over the list using index

```
range(0, 10)  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
[2, 3, 4, 5, 6, 7]  
[2, 5, 8, 11, 14, 17]
```

```
for i in range(len(genre)):  
    print(f"I like {genre[i]} music")
```

I like pop music

I like rock music

I like jazz music

CONTROL FLOW : LOOPING (FOR)

For Loop with Else

Example:

```
digits = [0, 1, 5]
for i in digits:
    print(i)
else:
    print("No items left.")
```

0
1
5
No items left.

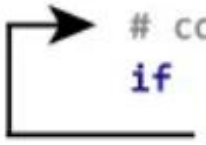
For Loop with Continue

Example:

```
for val in "string":
    if val == "i":
        continue
    print(val)
print("The end")
```

s
t
r
i
n
g
The end

```
for var in sequence:
    # codes inside for loop
    if condition:
        continue
    # codes inside for loop
# codes outside for loop
```



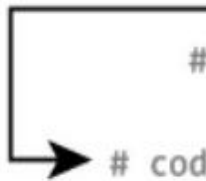
For Loop with Break

Example:

```
for val in "string":
    if val == "i":
        break
    print(val)
print("The end")
```

s
t
r
The end

```
for var in sequence:
    # codes inside for loop
    if condition:
        break
    # codes inside for loop
# codes outside for loop
```



CONTROL FLOW : LOOPING (FOR)

Comprehension, shorthand, and lambda function (anonymous functions are important for working with SPARK. Comprehensions topics are including items listed in this page onward.

LIST COMPREHENSION BASIC

Loop List

```
result = []  
nums = [12, 10, 13, 24, 17]  
for num in nums:  
    result.append(num + 1)  
print(result)  
Output: [13, 11, 14, 25, 18]
```

List Comprehension

```
result = [num+1 for num in nums]  
print(result)  
Output: [13, 11, 14, 25, 18]
```

List Comprehension with Range

List Comprehension - Nested Loop

CONTROL FLOW : LOOPING (FOR)

CONDITIONAL IN LIST COMPREHENSION

If with List Comprehension

Nested If with List Comprehension

If Else with List Comprehension

CONTROL FLOW : LOOPING (FOR)

DICTIONARY COMPREHENSION

GENERATORS

Generator Expression

Generator vs. List Comprehension

Printing Generators Value

Conditional in Generators

ITERATORS

Python's iterator object must implement two special methods, `__iter__()` and `__next__()`, which are collectively called the iterator protocol. An object is called iterable if we can implement iterator on to it. Lists, tuples, strings etc. are iterable.

Iteration of Iterables: next()

```
word = 'Data'  
it = iter(word)  
next(it)  
next(it)  
next(it)
```

't'

Iteration of Iterables: next()

```
word = 'Data'  
it = iter(word)  
print(*it)  
print(*it)
```

D a t a

ITERATORS

Iterators topics are including below:

Iteration of Dictionary

Iteration of A File

Enumerate()

Enumerate() and Unpack

Zip()

Zip() and Unpack

Zip() and Asterix (*)

FUNCTIONS

Python Built-in Functions (Examples)

Function	Example	Result
max()	points = [75, 67, 89, 70, 82, 65] print(max(points))	89
min()	points = [75, 67, 89, 70, 82, 65] print(min(points))	65
round(number, ndigit)	print(round(3.786, 2))	3.79

Python Built-in Methods (Examples)

Function	Example	Result
index()	fruits = ['apple', 'mango', 'orange'] print(fruits.index('orange'))	2
count()	fruits = ['apple', 'mango', 'orange'] print(fruits.count('orange'))	1
replace()	transport = 'bicylce' print(transport.replace('bi', 'tri'))	tricylce

Note: method is function owned by an object

FUNCTIONS

Python User Defined Functions (Examples)

Single argument

```
def square(value):  
    result = value ** 2  
    return result  
print(square(5))  
Output: 25
```

Return multiple values (tuple)

```
def squares(value1,  
value2):  
    result1 = value1 ** 2  
    result2 = value2 ** 2  
    return (result1,  
result2)  
print(squares(5,7))  
Output: (35, 49)
```

Multiple arguments

```
def area(length, width):  
    result = length * width  
    return result  
print(area(3, 8))  
Output: 24
```

Nested function

```
def mod2plus3(x1,x2,x3):  
    def calc(x):  
        return x%2 + 3  
    return (calc(x1), calc(x2),  
calc(x3))  
print(mod2plus3(1,2,3))  
Output: (4, 3, 4)
```

Lambda function

```
raise_to_power = lambda x, y: x ** y  
print(raise_to_power(3,2))  
Output: 9
```

*args Flexible Arguments

```
def OddEvent(*args):  
    result = ()  
    for arg in args:  
        if arg%2 == 0:  
            result = result + ('Event',)  
        else:  
            result = result + ('Odd',)  
    return result  
print(OddEvent(1,2,4,7))  
Output: ('Odd', 'Event', 'Event', 'Odd')
```

FUNCTIONS

Python User Defined Functions (Examples)

Function Generator

```
def num_sequence(n):  
    i = 0  
    while i < n:  
        yield i  
        i += 1  
result = num_sequence(4)  
print(type(result))  
for item in result:  
    print(item)
```

Output:
<class 'generator'>
0
1
2
3

Global scope

```
new_value = 100  
def square(value):  
    new_value = value ** 2  
    return new_value  
print(square(3))  
Output: 9
```

```
def sum(value):  
    result = new_value + value  
    return result  
print(sum(25))
```

Output: 125

**kwargs Argument

```
def greet(**kwargs):  
    """This function greets a person with the full  
    name."""  
    print("Hello", kwargs["first_name"] + ', ' +  
          kwargs["last_name"])  
greet(first_name="John", last_name="Smith")
```

Output: 'Hello John Smith'

OBJECT ORIENTED PROGRAMMING (OOP)

Python is an object-oriented programming (OOP) language, where the main emphasis is on the objects.

An object is a collection of data (variables) and methods (functions) that work on that data. Similarly, a class is the blueprint of that object. Objects are also called instances of classes and the process of creating these objects is called instantiate.

An object has two characteristics:

1. Attributes (variables)
2. Behavior (methods/functions)

The concept of OOP in Python focuses on creating reusable code. This concept is also known as **DRY** (Don't Repeat Yourself).



OBJECT ORIENTED PROGRAMMING (OOP)

Instantion of class with attributes into objects

class Car:

class attribute

type = 'sedan'

instance attribute

def __init__(self, brand, speed):

self.brand = brand

self.speed = speed

Output:

Bob drives a sedan

Bill drives a sedan

Bob's Honda car running at 150 km/h.

Bill's Toyota car running at 130 km/h.

instantiate the Car class

honda = Car('Honda', 150)

toyota = Car('Toyota', 130)

access the class attributes

print(f'Bob drives a {honda.__class__.type}')

print(f'Bill drives a {toyota.__class__.type}')

access the instance attributes

print(f'Bob\'s {honda.brand} car running at {honda.speed} km/h.')

print(f'Bill\'s {toyota.brand} car running at {toyota.speed} km/h.')

OBJECT ORIENTED PROGRAMMING (OOP)

Instantion of class with attributes & method into objects

class Parrot:

instance attributes

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

instance method

```
def sing(self, song):  
    return f"{self.name} sings {song}."  
def dance(self):  
    return f"{self.name} is now dancing."
```

instantiate the object

```
blu = Parrot("Blu", 10)
```

call our instance methods

```
print(blu.sing("Happy"))  
print(blu.dance())
```

Output:

Blu sings Happy.

Blu is now dancing.

OBJECT ORIENTED PROGRAMMING (OOP)

Inheritance

Inheritance is a way of creating a new class to use the details of an existing class without modifying it. The newly created class is a derived class (or child class). Similarly, the existing class is the base class (or parent class).

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")
```

#instantiate child class

```
peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Output:

*Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster*

OBJECT ORIENTED PROGRAMMING (OOP)

Encapsulation

By using OOP in Python, we can restrict access to methods and variables. This preventing our data from being modified directly, also known as encapsulation. In Python, we can denote private attribute using underscore as prefix i.e. `_` or `__`

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

#instantiate class Computer

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

Output:

Selling Price: 900

Selling Price: 900 #no change

Selling Price: 1000 #changed by setMaxPrice

OBJECT ORIENTED PROGRAMMING (OOP)

Polymorphism

Polymorphism is OOP's ability to use a **common interface** for a variety of applications.

```
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()
```

```
def swimming_test(bird):
    bird.swim()
```

```
#instantiate objects
blu = Parrot()
peggy = Penguin()
```

```
# passing the object
flying_test(blu)
flying_test(peggy)
swimming_test(blu)
swimming_test(peggy)
```

Output:

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

VERSION CONTROL SYSTEM (VCS)

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes of source codes over time.

Local Version Control Systems

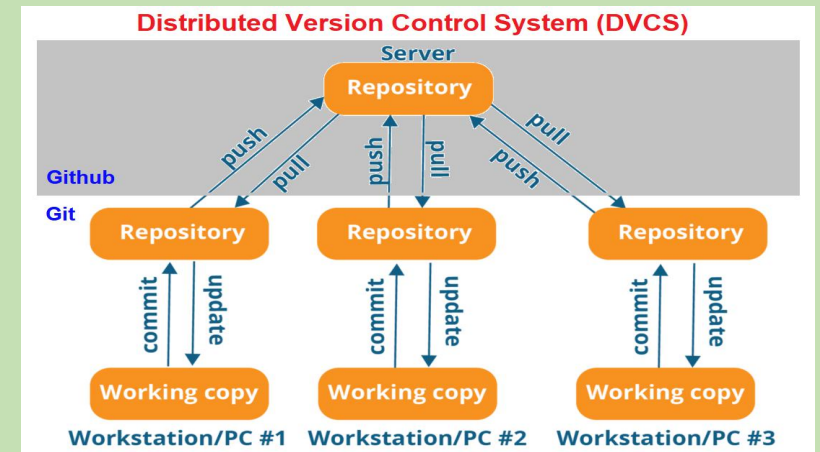
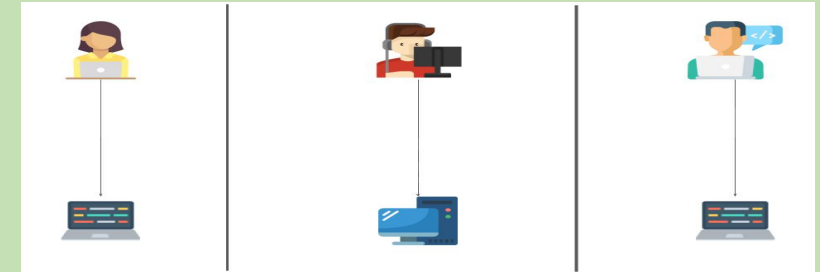
A VCS but without a remote repository (repo). You manage and version all the files only within your local system. There is no remote server in this scenario. All the changes are recorded in a local database.

Centralized Version Control Systems (CVCS)

A central repo shared with all the developers, and everyone gets their own working copy. Whenever you commit, the changes get reflected directly in the repo.

Distributed Version Control Systems (DVCS)

A local copy of the repo for every developer on their computers. They can make whatever changes they want and commit without affecting the remote repo directly. They first commit in their local repo and then push the changes to the remote repo. This is the type used majorly today including by Git & Github.



VERSION CONTROL SYSTEM : GIT

Git is one of the most popular VCS tools in use today. Git is a Distributed VCS, a category known as DVCS. Git is free and open source.

Snapshot

Git records all files at any given time to track file changes. We can access all changes that have been made.

Commit

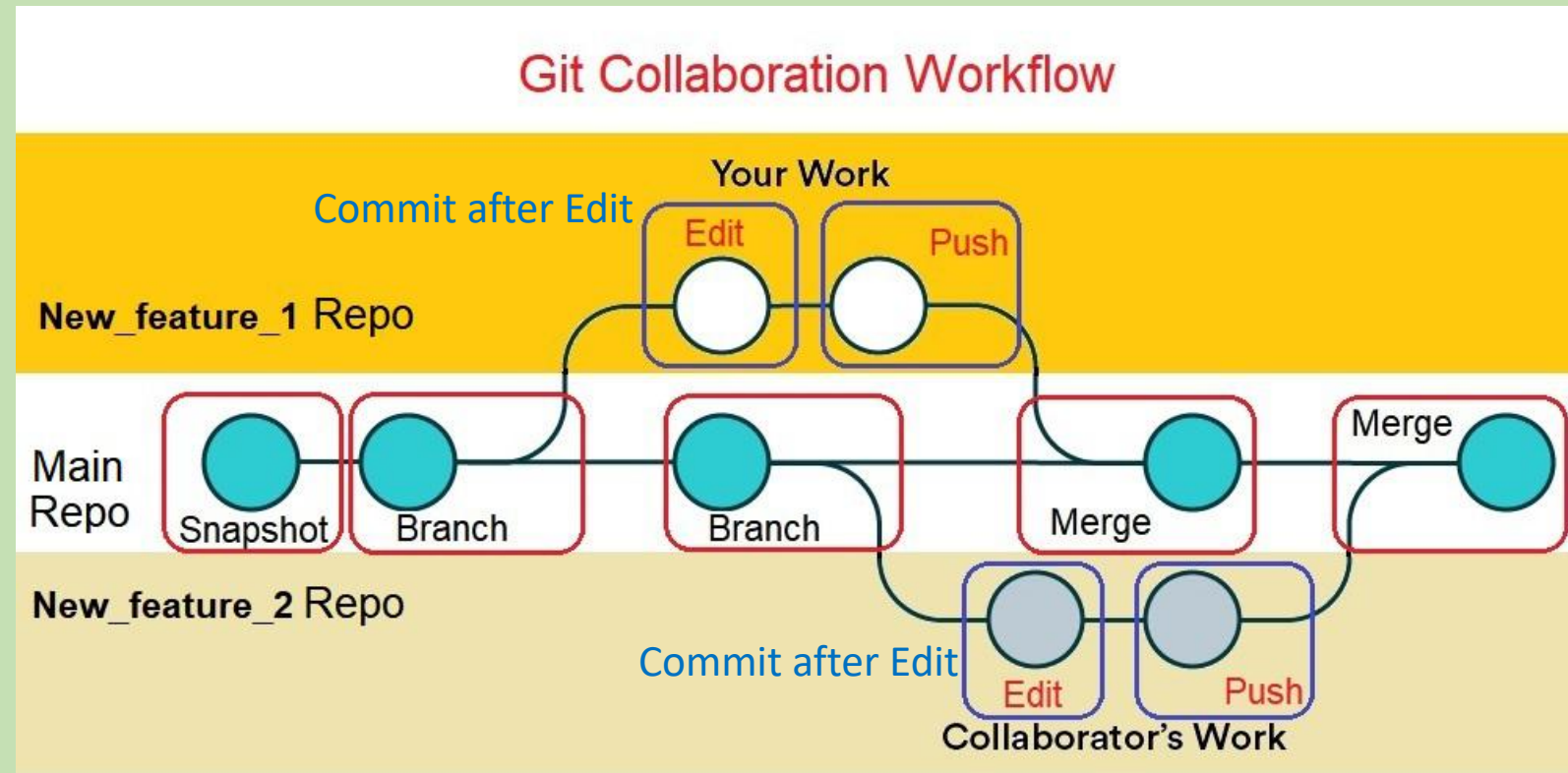
Similar to saving a file that's been **edited**, a commit records changes to one or more files in your branch. Git assigns each commit a unique ID, called a SHA or hash, that identifies: The specific changes. A commit is a snapshot in time.

Repo (Repository)

A collection of many files and their changes, also the place where all commits are stored.

Branch

A version of your repository, or in other words, an independent line of development.



VERSION CONTROL SYSTEM : GIT

Git is useful for:

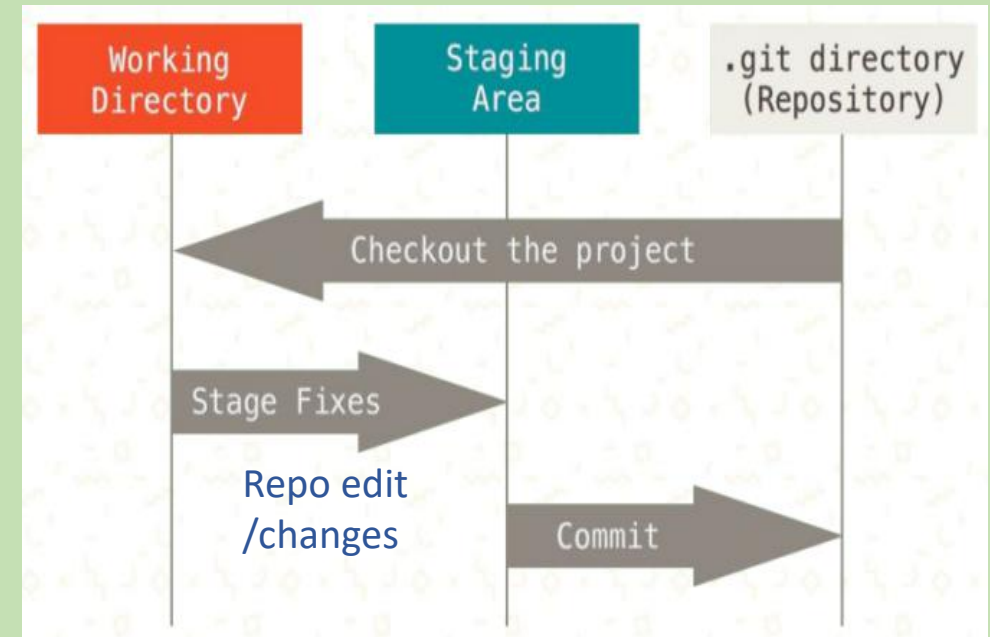
- * Track code changes, and possibility to revert and undo changes back to previous version
- * Keep track of who made the changes
- * Collaboration in working on code
- * Easy to solve code conflict

What can be done in Git:

- * Manage projects by creating Repositories
- * Clone projects to work a copy of the code on the local system
- * Control and track changes with Staging and Committing
- * Branch and Merge to allow work on different parts and versions of projects different
- * Pull latest version of projects to local copy - Push local update to main projects

Commit Steps:

1. **Modified:** make changes to the file but not yet committed
2. **Staged:** mark which files have been changed and want to commit
3. **Committed:** save changes in git



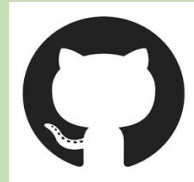
Git Areas:

1. **Working dir:** a place where we do changes to files
2. **Staging area:** contains files that have been marked for commit
3. **.git dir:** contains changes that have been committed

VERSION CONTROL SYSTEM : GITHUB

GitHub is a product that allows us to host Git projects on remote servers or the cloud.

GitHub is not Git. GitHub is a hosting service. Other companies that offer hosting services similar to GitHub are **Bitbucket** and **GitLab**.



We need to have Github account to work with the service. When we create NEW repository, to connect our local system with the remote system (Github), authentication is required.

HTTPS connection

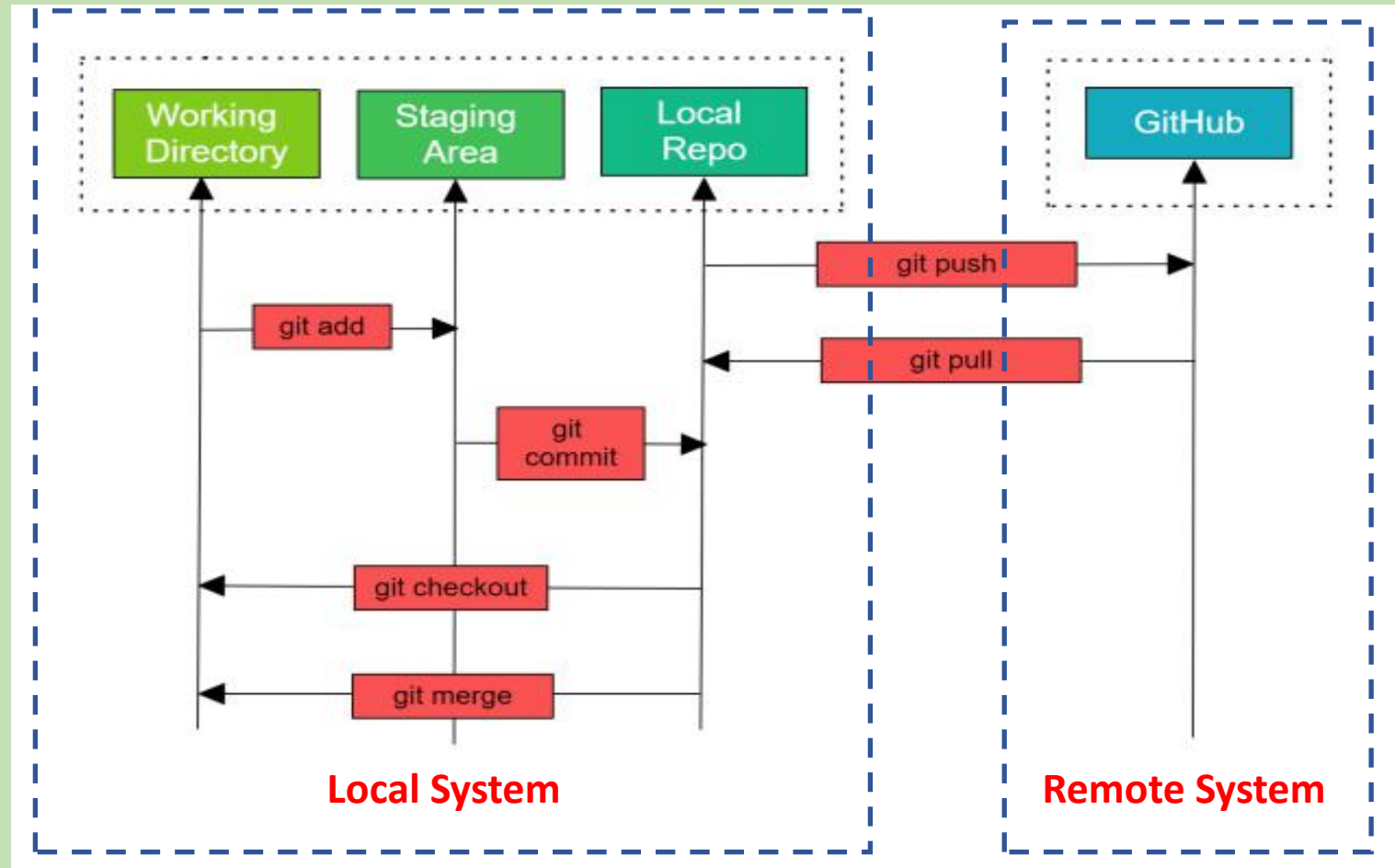
Usually for public repo use this connection. Authentication window will pop up. Login name and password is required to be input.

SSH connection

Usually used for private connection. SSH key is required by Github. We can get SSH keys in local system by running key-gen.

VERSION CONTROL SYSTEM : GIT & GITHUB

Git & Github Workflow



VERSION CONTROL SYSTEM : GIT & GITHUB

Basic Git Commands

Command	Description
git init <directory_name>	Creates an empty Git repository in specific directory.
git clone <link_repo>	The cloned repo is located in <link_repo> to local computer. Original repo can be located in the local file system or on remote machine via HTTP or SSH.
git config -- global user.name	Determine the name of the author who will used for all commits in your name
git config -- global user.email	Determine the user email of the author who will used for all commits in your name
git add <directory_name>	Save on stage all changes <directory_name> for commit next. Change <directory_name> with <file_name> to change to a specific file. Put "." as <directory name> to add all files and folders within the directory.
git commit -m "write message"	Commit saved snapshot on stage, write message as commit message.
git status	Lists which files are in staged, unstaged, and not traceable.

VERSION CONTROL SYSTEM : GIT & GITHUB

Basic Git Commands

Command	Description
git log	Show entire commit history.
git diff	Show changes without label (unstaged) between the index in local and working directory
git push <remote-name> <branch name>	send local commits to repository branch remote (remote repo).
git checkout -b <branch-name>	creates a new branch and switches to new branches.
git remote -v	View all remote repos in the account certain
git pull	merge commits to directory local from the remote repository.
git branch -d <branch-name>	Delete a branch
git merge <branch-name>	After resolving merge conflict, this command merges branch selected to the current branch

THANK YOU