
INTRODUCTION TO
PYTHON

Agenda

In this module, we will look at the following topics:

- ✓ What is Python ?
- ✓ Why Python ?
- ✓ Python Applications & Trends
- ✓ Python Basics
- ✓ Python Data Types
  - Immutable: Numbers, Strings, Tuples
  - Mutable : Lists, Dictionaries, Sets
- ✓ Sequences & Collections
- ✓ Python Operators
- ✓ Conditional Statements & Loops
- ✓ Command-line Arguments
- ✓ File Operations

What is Python?

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.

Why Python?

1

Simple and Easy to Learn

Python is simple and easy to learn, understand and write than most other programming languages.

Why Python?

1. **Simple and Easy to Learn**

2. **Open Source**

   Python is open-source, which means one can freely distribute the copies of this software, make modifications to its source code etc.

Why Python?

1. **Simple and Easy to Learn**
2. **Open Source**
3. **High-Level Language**

   Abstracts low-level details such as memory allocation etc, so that developers need not bother about such things while writing programs using python.

Why Python?

1. **Simple and Easy to Learn**
2. **Open Source**
3. **High-Level Language**
4. **Multi-paradigm**

   Supports both objected-oriented and procedure-oriented programming.

Why Python?

1. **Simple and Easy to Learn**
2. **Open Source**
3. **High-Level Language**
4. **Multi-paradigm**
5. **Portable**

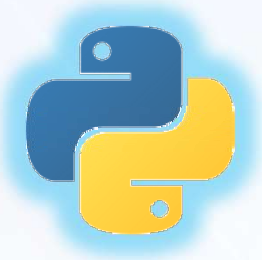
   Supported by many different platforms like Windows, Linus, Mac, FreeBSD, Solaris etc.

Why Python?

1. **Simple and Easy to Learn**
2. **Open Source**
3. **High-Level Language**
4. **Multi-paradigm**
5. **Portable**
6. **Extensible**

Python can invoke C & C++ libraries, can be integrated with Java and .Net components

Python Applications



Data Analysis

Pandas

Web Development

Django integrated

Testing

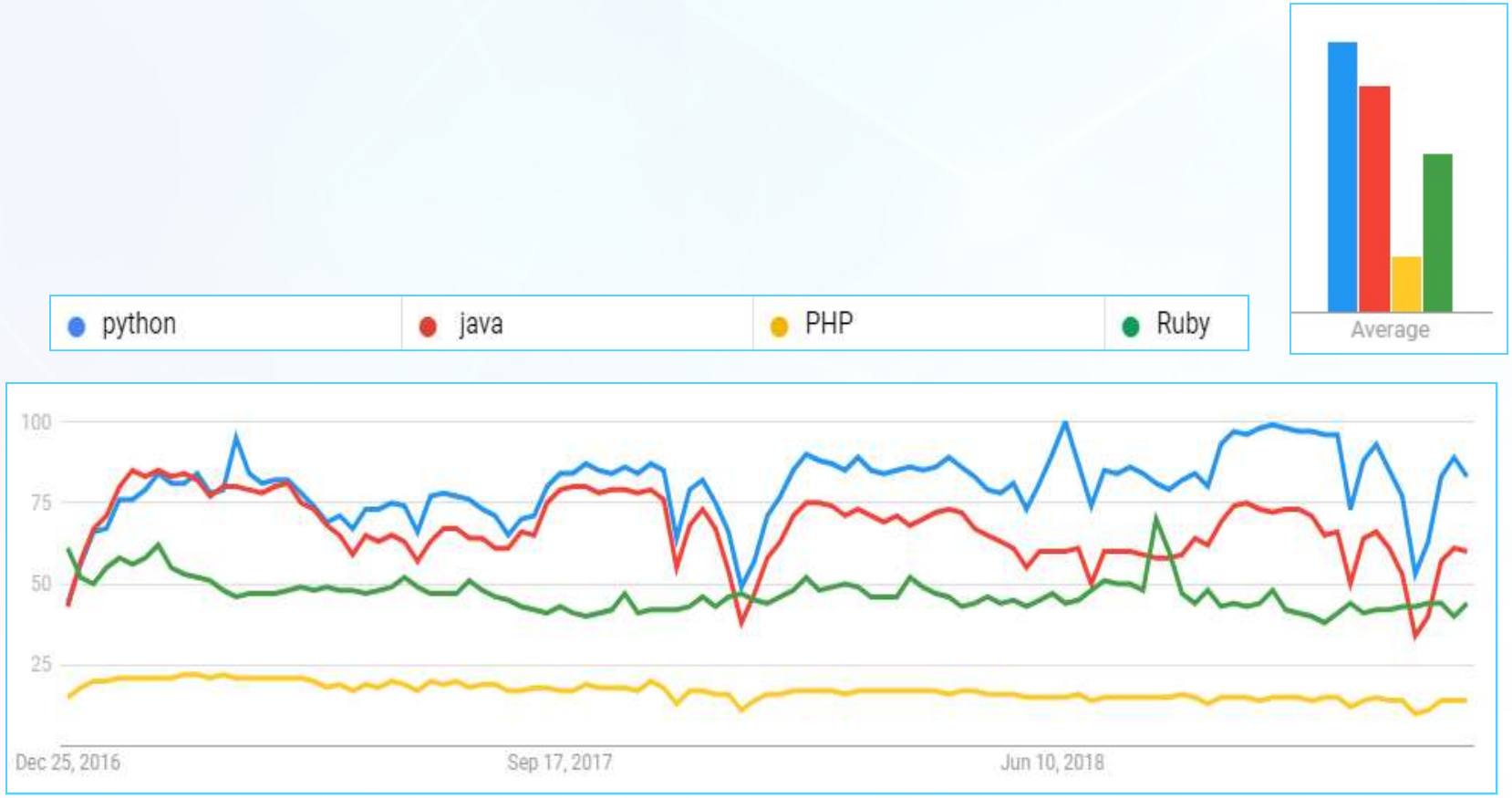
Selinium integrated

Web Scraping

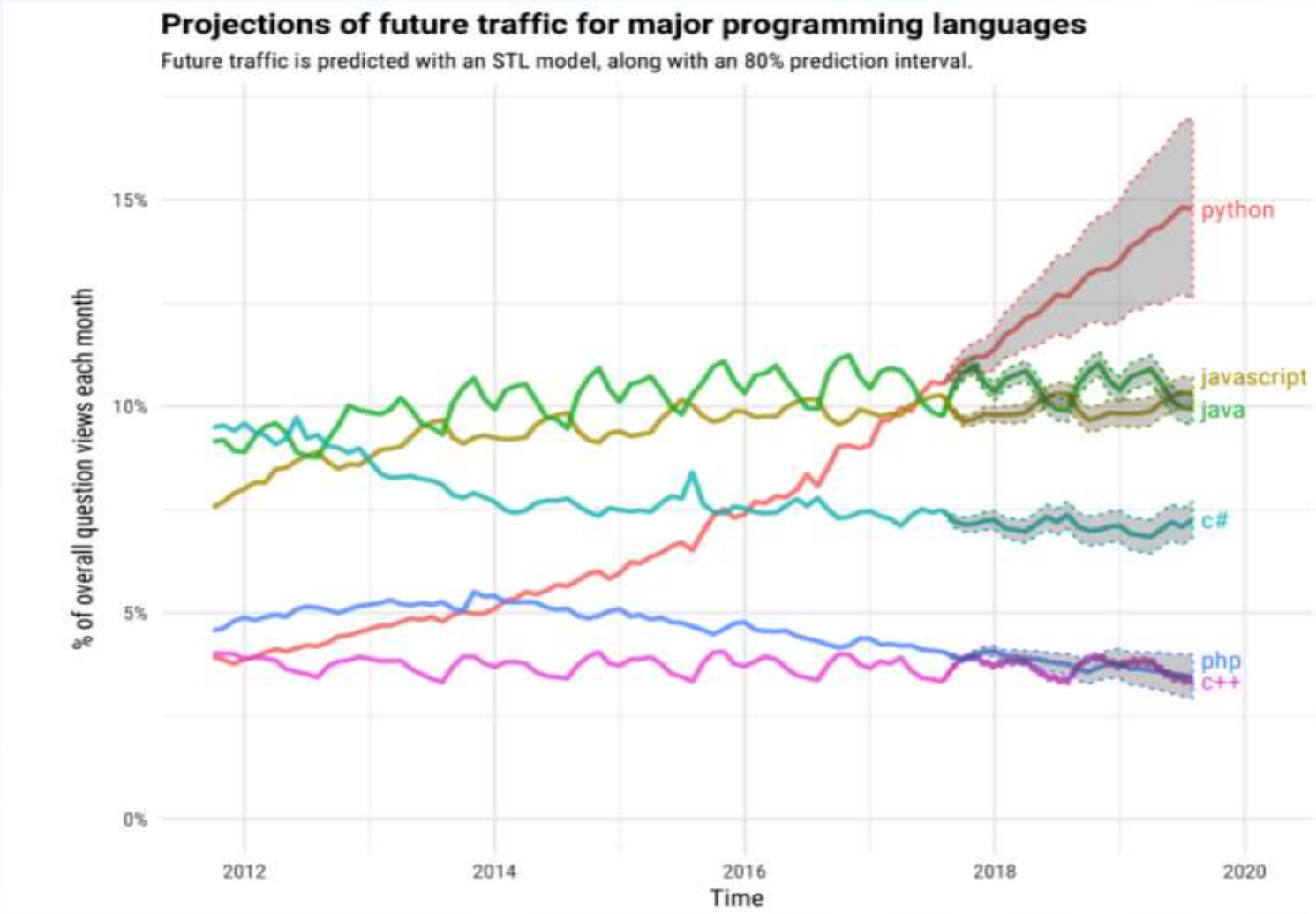
...

...

Python on Google Trends



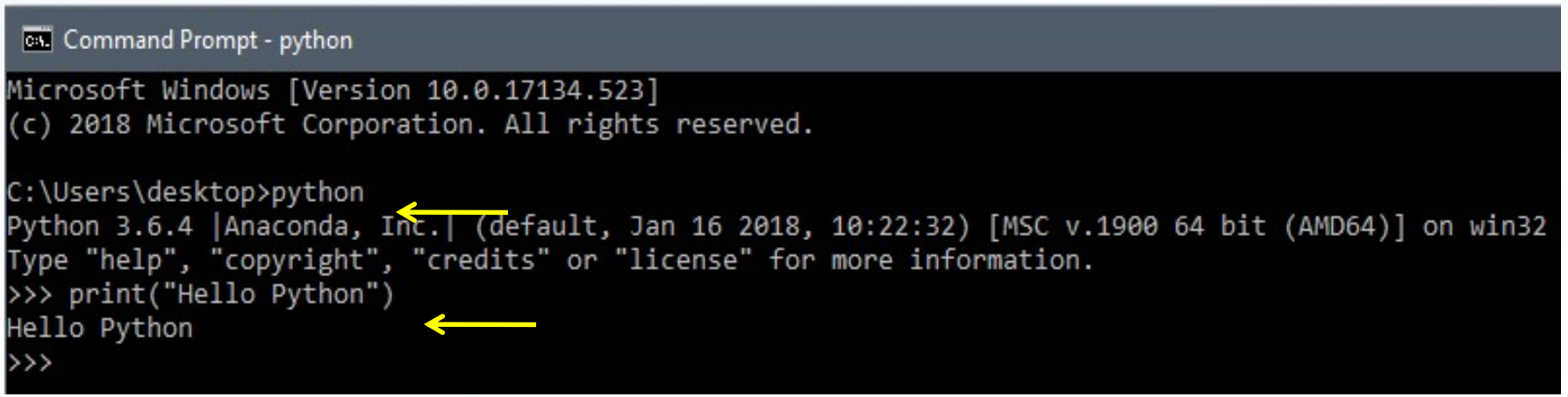
Python Trends



Python Interpreter

Python Interpreter

- reads and executes python code line-by-line
- type “python” at the command program to launch
- You can run python commands from the shell



Python Basics

Comments

- Single Line Comments: **Use #**
- Bulk Comment : **Enclose in “““ or “**

```
# This is single line comments
# use it to describe your code

"""
This is bulk comment
Any code enclosed between triple-quotes
is treated as comment block
"""


'''
You can also use triple-single-quotes
for bulk comments
'''
```

Identifiers

Identifier

- Name used to identify a variable, function, class etc.
- Should start with an alphabet or _ followed by alphanumerics and underscores
- Special characters such as $, @ & % are not allowed within identifiers.
- Python is case sensitive

```
a = 10
print(a)

a1 = 'hello'
1a = 'hello again'

$x = "does it work?"
print($x)

A = 100
print(A)
```

→ Error: 1a is not a valid identifier

→ Error: $ is not allowed in an identifier

→ ***A*** and ***a*** are different

Identifier Naming Conventions

Class names start with an upper-case letter. All other identifiers start with a lower case letter

Private identifiers start with a single leading underscore (eg: _age)

Strongly private identifiers start with a two leading underscore (eg: __age)

Language defined special names contain two leading as well as trailing underscores (eg: __name__)

Indentation

Braces are not used to indicate blocks of code.

Blocks of code are denoted by **strictly enforced line indentation.**

Number of spaces to use is not fixed, but all statements within the block should follow the same indentation.

```
def convert_age(age):
    if(age >= 0 and age <= 10):
        return 'Child'
    elif(age <= 25):
        return 'Young'
    elif(age <= 50):
        return 'Middle'
    else:
        return 'Old'
```

Multiline-Statements

To write multi-line statements you have to use the line continuation character (\ ) to denote that the line should continue.

```
total = 10 +
        20 +
        30

print(total)
```

→ Invalid code

```
total = 10 + \
        20 + \
        30

print(total)
```

→ Valid code

Quotes

- Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals.
- The triple quotes are used to span the string across multiple lines.

```
word = 'word'

sentence = "This is a sentence."

paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""

print(paragraph)
```

Multiple Statements on a Single Line

- The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block.

```
name = 'adam'; age = 20; print(name, age)
```

Variable Assignment

Assigning a value to a variable

Assigning values to multiple variables at the same time

```
name = "John"    # A string
age = 25         # An integer assignment
height = 5.8     # A floating point


name, age, height = "John", 25, 5.8


a = b = c = 100
```

Reading from Keyboard

Use **`input`** method to read user input from the standard input device (i.e Keyboard). The input is always returned as a `string`.

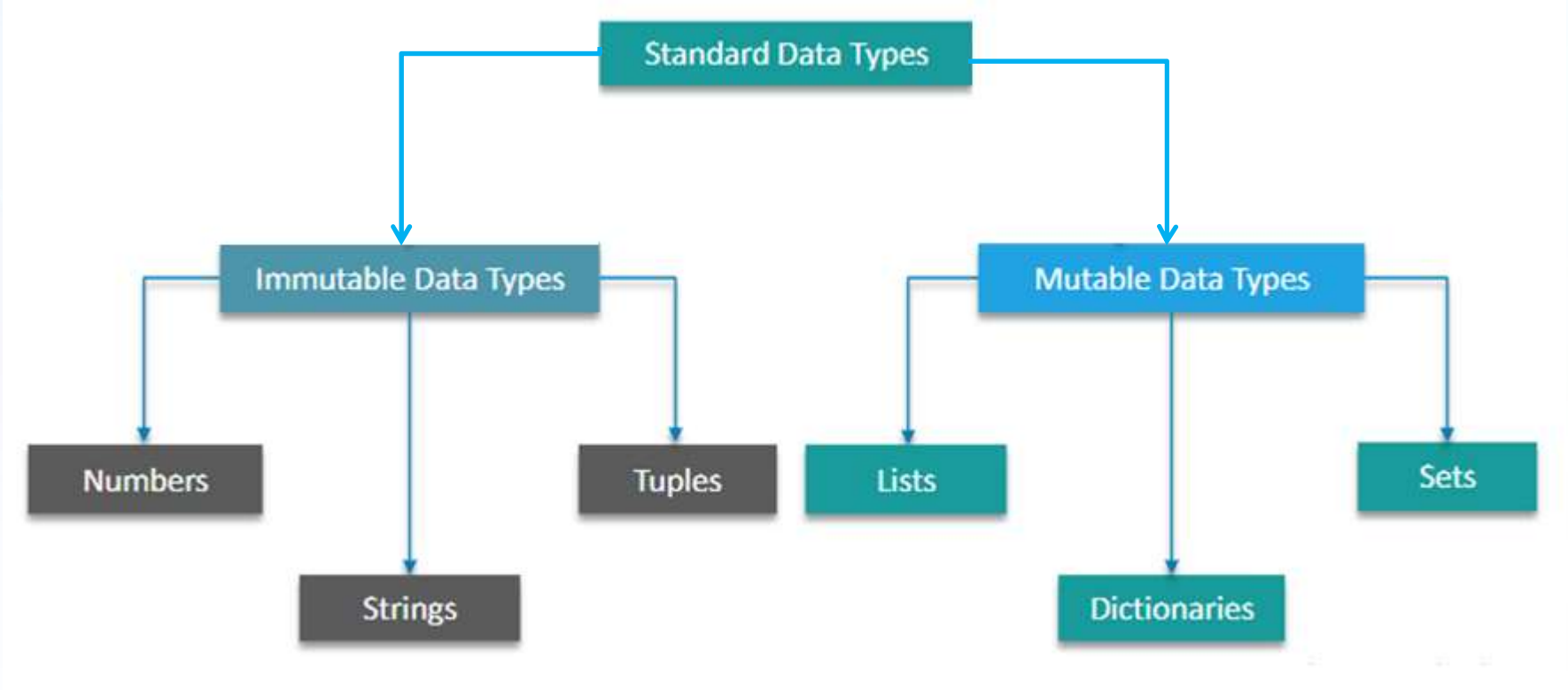
```
name = input("What's your name? ")
print("Hello " + name + "!")

age = input("Your age? ")
print( name + " is " + str(age) + " years old")

cities = input("Largest cities in India: ")
print(cities, type(cities))

cities = eval(input("Largest cities in India: "))
print(cities, type(cities))

population = input("Population of India? ")
print(population, type(population))

population = int(input("Population of
print(population, type(population))
```

```
What's your name? Raju
Hello Raju!

Your age? 40
Raju is 40 years old
```

```
Largest cities in India: ["New Delhi", "Mumbai", "Chennai"]
["New Delhi", "Mumbai", "Chennai"] <class 'str'>

Largest cities in India: ["New Delhi", "Mumbai", "Chennai"]
['New Delhi', 'Mumbai', 'Chennai'] <class 'list'>

Population of India? 1200000000
1200000000 <class 'str'>

Population of India? 1200000000
1200000000 <class 'int'>
```

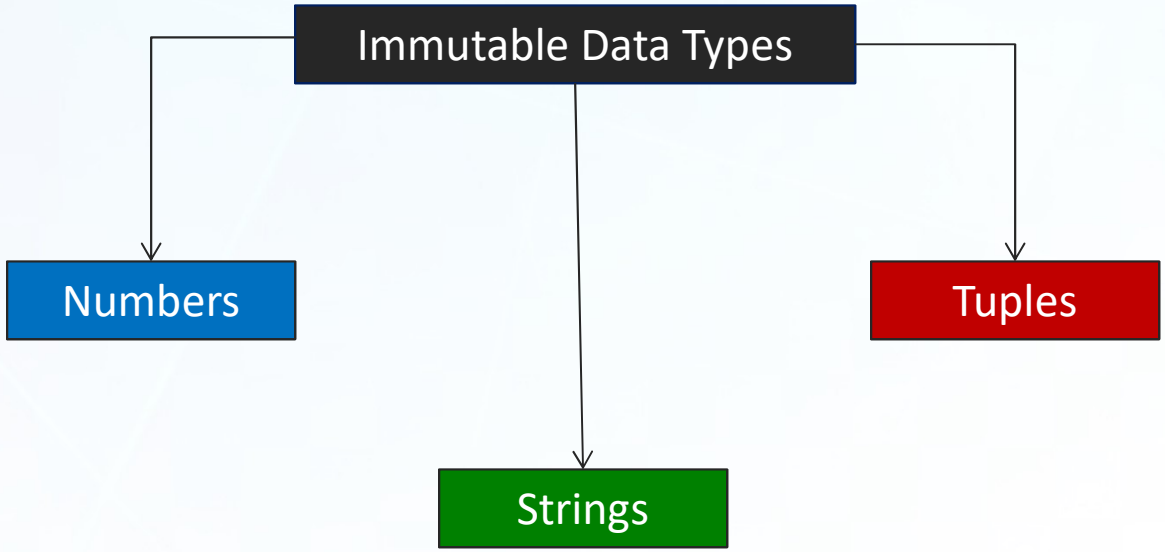
Python Data Types

Python Data Types



Immutable Data Types

Immutable objects doesn't allow modification after creation



Understanding Immutability

```
a = 10
```

When a variable is created, the value is stored in a memory location identified by an id. `id(a)` command will give you the id of the memory location where the value of `a` is stored.

Understanding Immutability

```
a = 10
```

```
a = a + 1
```

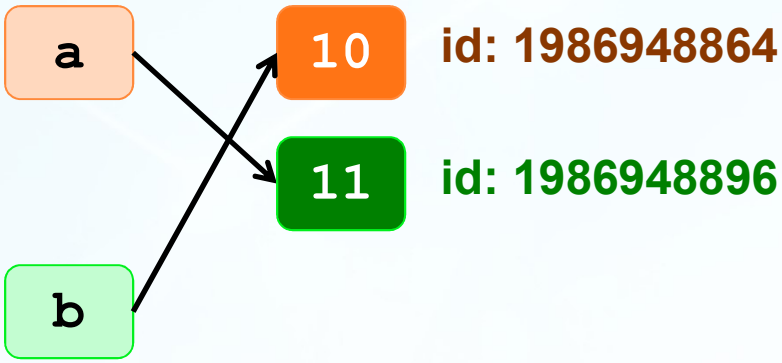
Integer 10 is immutable, so the value in the memory location where the integer 10 is stored can not be changed.

Understanding Immutability

```
a = 10
```

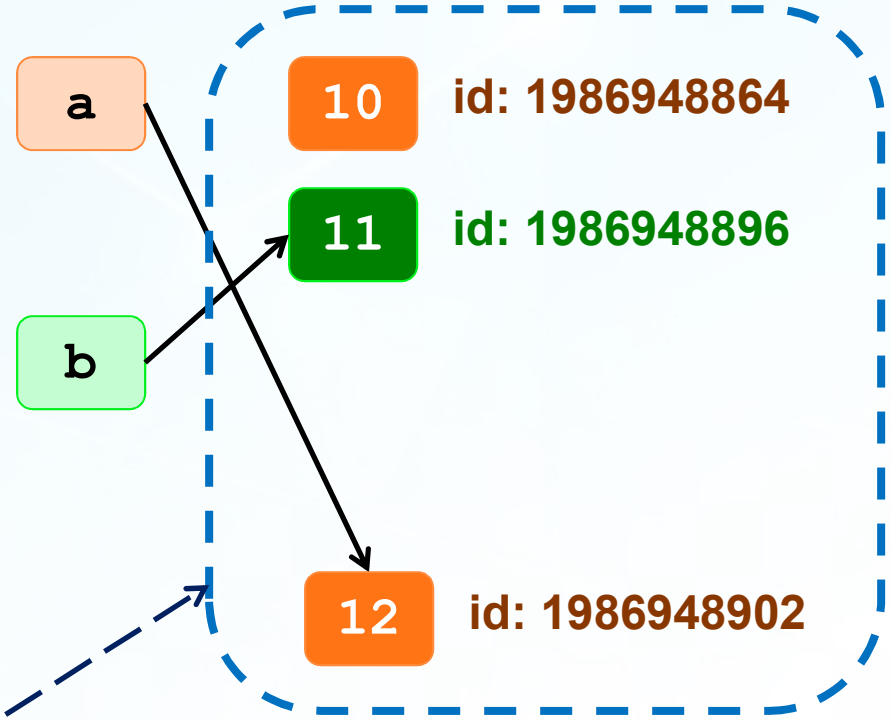
```
a = a + 1
```

```
b = 10
```



Understanding Immutability

```
a = 10
a = a + 1
b = 10
b = 11
a = a + 1
```



These integers are immutable. Hence, whenever a new value is assigned, a new memory location is created.

Numeric

Numbers

Strings

Tuples

- Python supports three numeric types

Integers `a = 5`

Floating Points `b = 6.25`

Complex Numbers `c = 2 + 3j`

```
x = 5; print (x, type(x))
y = 6.25; print (y, type(y))
z = 2 + 3j; print (z, type(z))
print(y * z)
```

Python Code

```
5 <class 'int'>
6.25 <class 'float'>
(2+3j) <class 'complex'>
(12.5+18.75j)
```

Output

Strings

Numbers

Strings

Tuples

```
str1 = 'Welcome to Python.'
str2 = "Machine Learning is cool."

print(str1, str2)
print(str1 + str2)
print(str1 * 3)
print(f"Length of '{str1}', is {len(str1)}" )

# string slicing
print(str1[0], str1[1], str1[5])
print(str1[1:5])
print(str1[:15])
print(str1[3:])
```

You can use both single and double quotes for strings

String Slicing

```
W e m
elco
Welcome to Pyth
come to Python.
```

```
Welcome to Python. Machine Learning is cool.
Welcome to Python.Machine Learning is cool.
Welcome to Python.Welcome to Python.Welcome to Python.
Length of 'Welcome to Python.', is 18
```

Tuples

Numbers

Strings

Tuples

Tuples store a list of values of different data-types separated by comma and enclosed in parenthesis.

```
t1 = (1, 2.5, 'python', 2.0, 1.23, 'hello', 4, 6, 7);
print(t1)

t2 = (11, 12.5, 'machine learning', t1);
print(t2)

print(t1[1], t1[-1], t2[-2], t2[3][0], t2[-1][-2])
```

```
2.5 7 machine learning 1 6
```

```
(11, 12.5, 'machine learning', (1, 2.5, 'python', 2.0,
1.23, 'hello', 4, 6, 7))
```

Tuple Slicing

```
t1 = (1, 2.5, 'python', 2.0, 1.23, 'hello', 4, 6, 7);
t2 = (11, 12.5, 'machine learning', t1);
print(t1[1:4])
print(t1[1:-1])
print(t2[-1][-2:])
print(t2[3][2:])
print(t1[0:len(t1)])
print(t1[0:len(t1):2])
print(t1[::-1])
print(t1[6:1:-2])
```

code

output

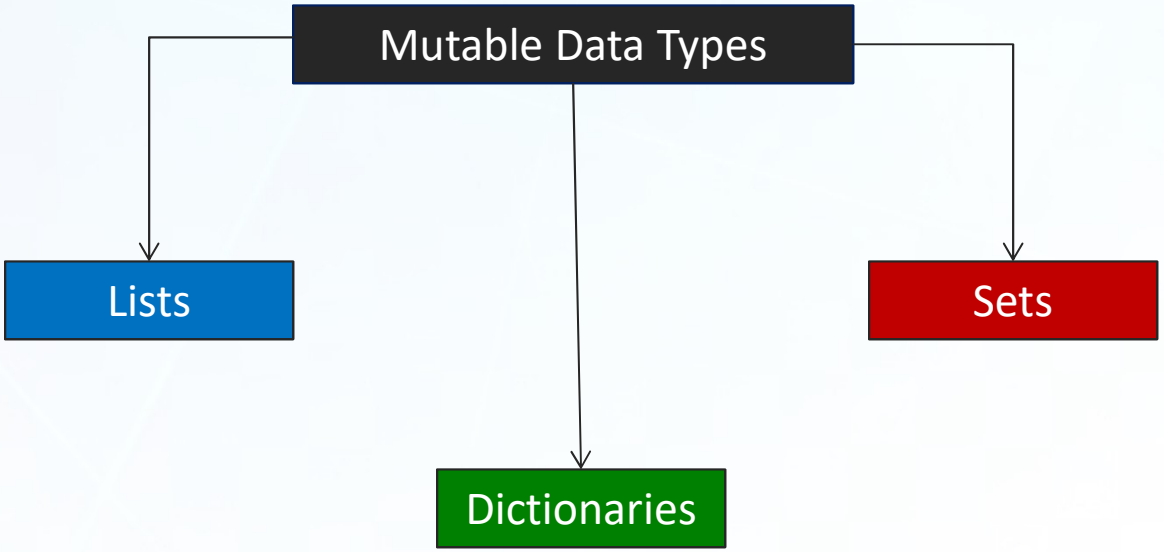
```
(2.5, 'python', 2.0)
(2.5, 'python', 2.0, 1.23, 'hello', 4, 6)
(6, 7)
('python', 2.0, 1.23, 'hello', 4, 6, 7)
(1, 2.5, 'python', 2.0, 1.23, 'hello', 4, 6, 7)
(1, 'python', 1.23, 4, 7)
(7, 6, 4, 'hello', 1.23, 2.0, 'python', 2.5, 1)
(4, 1.23, 'python')
```

a : b : c

- **a** : starting index
- **b** : ending index
- **c** : skip interval

Mutable Data Types

The contents of mutable objects can be changed after creation.



Lists

Lists

Dictionaries

Sets

- List is an ordered set of elements enclosed in square brackets

```
list = [22, "python", 20.5, 7, 4]
```

- Lists are mutable i.e. List object can be modified.

```
list[0] = 33
```

- Lists are slower than Tuples.

Lists

```
l1 = []
l2 = list()
l3 = [22, "python", 20.5]
print(l3, id(l3))
l3[0] = 33
print(l3, id(l3))
l3.extend([10.23, 'Bill'])
print(l3, id(l3))
l3.append(40)
print(l3, id(l3))
l3.insert(2, -10.4)
print(l3, id(l3))
l3.pop()
print(l3, id(l3))
l3.pop(4)
print(l3, id(l3))
l3.remove(7)
print(l3, id(l3))
```

code

Lists are mutable as shown here.

output

```
[22, 'python', 20.5, 7, 4] 1842119992008
[33, 'python', 20.5, 7, 4] 1842119992008
[33, 'python', 20.5, 7, 4, 10.23, 'Bill'] 1842119992008
[33, 'python', 20.5, 7, 4, 10.23, 'Bill', 40]
1842119992008
[33, 'python', -10.4, 20.5, 7, 4, 10.23, 'Bill', 40]
1842119992008
[33, 'python', -10.4, 20.5, 7, 4, 10.23, 'Bill']
1842119992008
[33, 'python', 20.5, 7, 4, 10.23, 'Bill'] 1842119992008
[33, 'python', 20.5, 4, 10.23, 'Bill'] 1842119992008
```

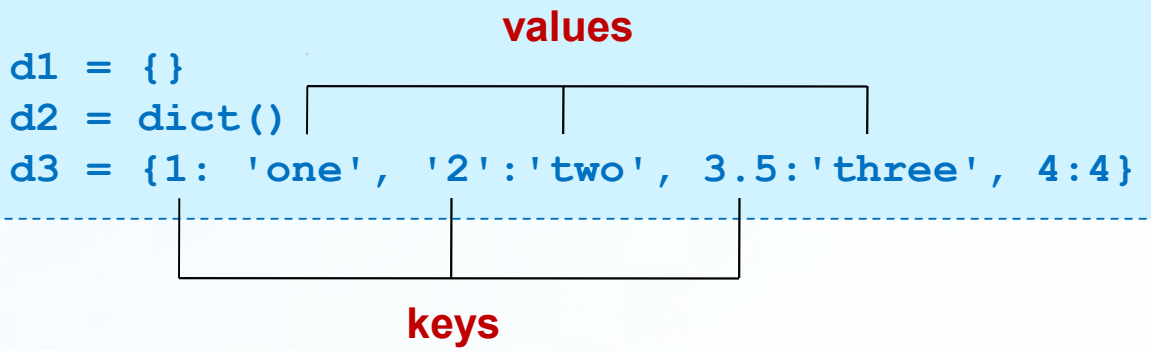
Dictionaries

Lists

Dictionaries

Sets

- Dictionaries contain key-value pairs delimited by colon (:)
- The collection of key-value pairs are enclosed in { .. }



Dictionaries

code

```
d1 = {1: 'one', '2':'two', 3.5:'three'}

print( d1, id(d1) )
print( d1.keys(), type(d1.keys()) )
print( d1.values(), type(d1.values()) )
print( d1.items(), type(d1.items()) )

d1['five'] = 5  # add/update an item
d1[1] = 1
print (d1, id(d1))

#delete using pop or del
d1.pop('2')
del d1[3.5]

print (d1, id(d1))
```

output

```
{1: 'one', '2': 'two', 3.5: 'three'} 1842120042536
dict_keys([1, '2', 3.5]) <class 'dict_keys'>
dict_values(['one', 'two', 'three']) <class 'dict_values'>
dict_items([(1, 'one'), ('2', 'two'), (3.5, 'three')]) <class 'dict_items'>
{1: 1, '2': 'two', 3.5: 'three', 'five': 5} 1842120042536
{1: 1, 'five': 5} 1842120042536
```

Sets

Lists

Dictionaries

Sets

- Sets are **unordered** collections of **unique** elements.
- Use {..} with elements separated by comma (,)

```
s = {1,2,3,4,4,3}
print(s)
```

```
{1, 2, 3, 4}
```

duplicate elements are removed from the set

- The simplest way to remove duplicate elements from a list is to convert it into a set.
- Because sets are *unordered* you can not fetch set value using index. You need to convert them into list/dictionary etc to access set elements.

Sets

code

```
l1 = [98,44,67,98,56,34,67,89,12,55,66,77,55,66,77]
s1 = set(l1)

s2 = {98,99,55,78,67,56,45,44}

print(s1)
print(s2)

s3 = s1 & s2    # intersection
s4 = s1 | s2    # union
s5 = s1 - s2    # difference
s6 = s2 - s1    # difference
s7 = s1.symmetric_difference(s2)

print(s3)
print(s4)
print(s5)
print(s6)
print(s7)
```

output

```
{98, 67, 34, 66, 44, 12, 77, 55, 56, 89}
{98, 67, 99, 44, 45, 78, 55, 56}
```

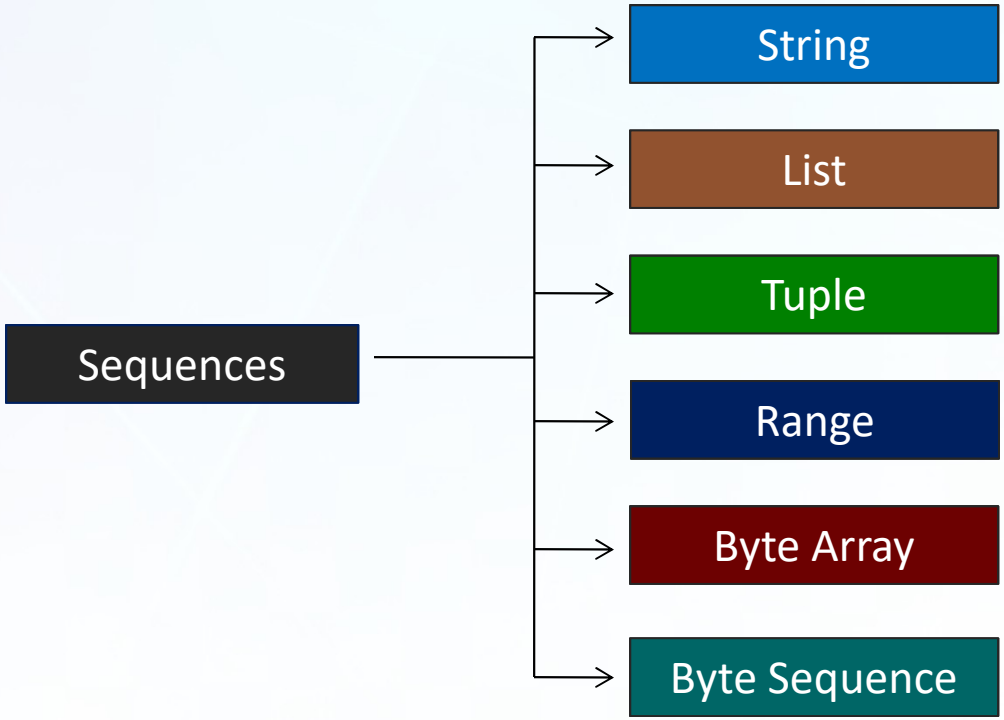
output

```
{98, 67, 44, 55, 56}
{98, 67, 34, 66, 99, 44, 12, 77, 45, 78, 55, 56, 89}
{34, 66, 12, 77, 89}
{99, 45, 78}
{34, 66, 99, 12, 45, 78, 77, 89}
```

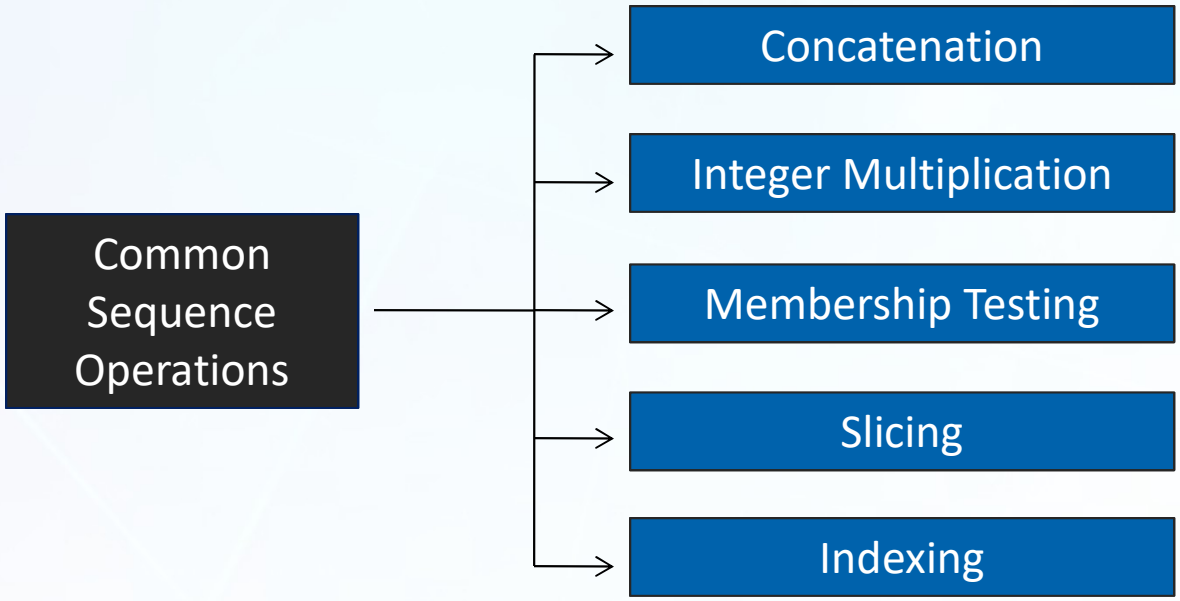
Sequences & Collections

Sequences

A **sequence** is a group of items with a deterministic ordering. The order in which we put them in is the order in which we get an item out from them.



Sequence Operations



Lists vs Tuples

Lists

- Mutable and slower (than Tuples)
- Use them when you have data that does not need random access and if values need to be changed

Tuples

- Immutable and faster
- Use them when you have data that need not be changed and require fast random access

Range object

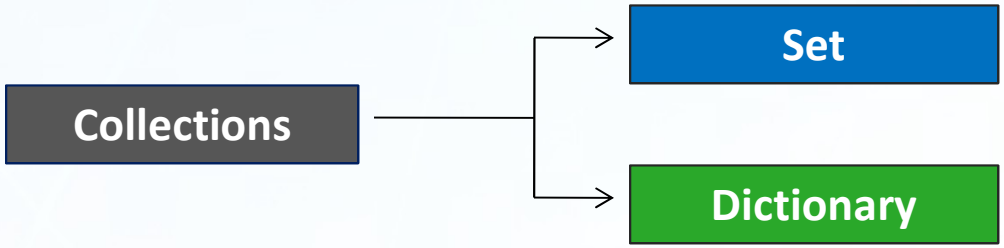
- `range` object lends us a range to iterate on; it gives us a list of numbers.

```
r1 = range(5)
print ( r1, type(r1) )

r2 = range(30, 0, -5)
print ( r2, type(r2) )

for i in r2:
    print (i, end = " ")
```

```
range(0, 5) <class 'range'>
range(30, 0, -5) <class 'range'>
30 25 20 15 10 5
```

Collections

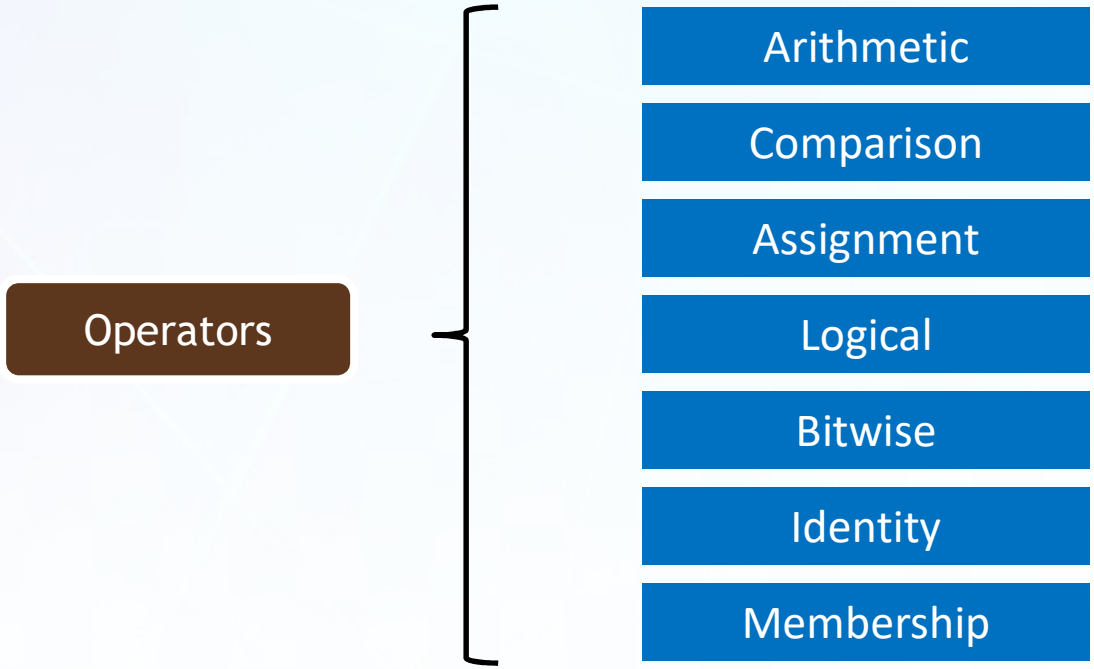
- Python **collection**, unlike a sequence, does not have a deterministic ordering. Examples include sets and dictionaries.
- In a collection, while ordering is arbitrary, physically, they do have an order.



Python Operators

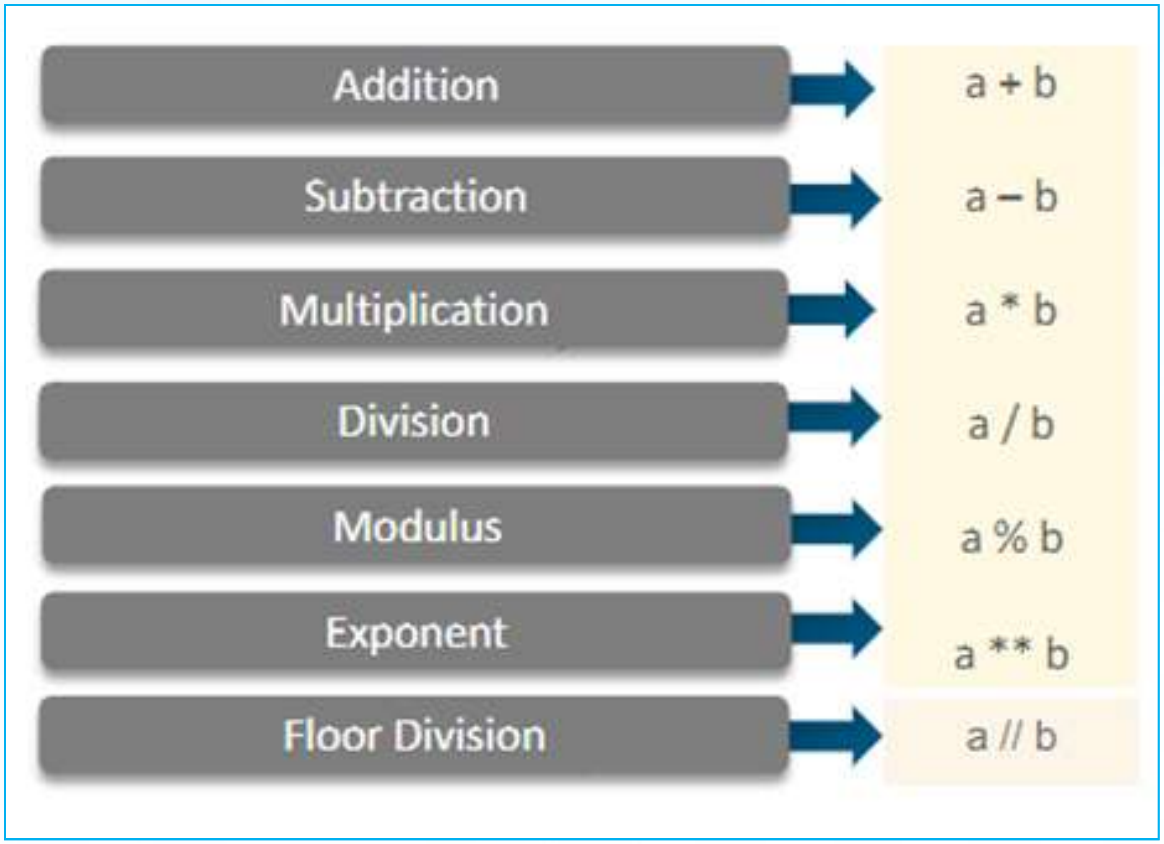
Operators

Operators are constructs that operate on the operands to give a value. For ex. in 6 * 2 = 12, 6 and 2 are called operands and * is an operator.



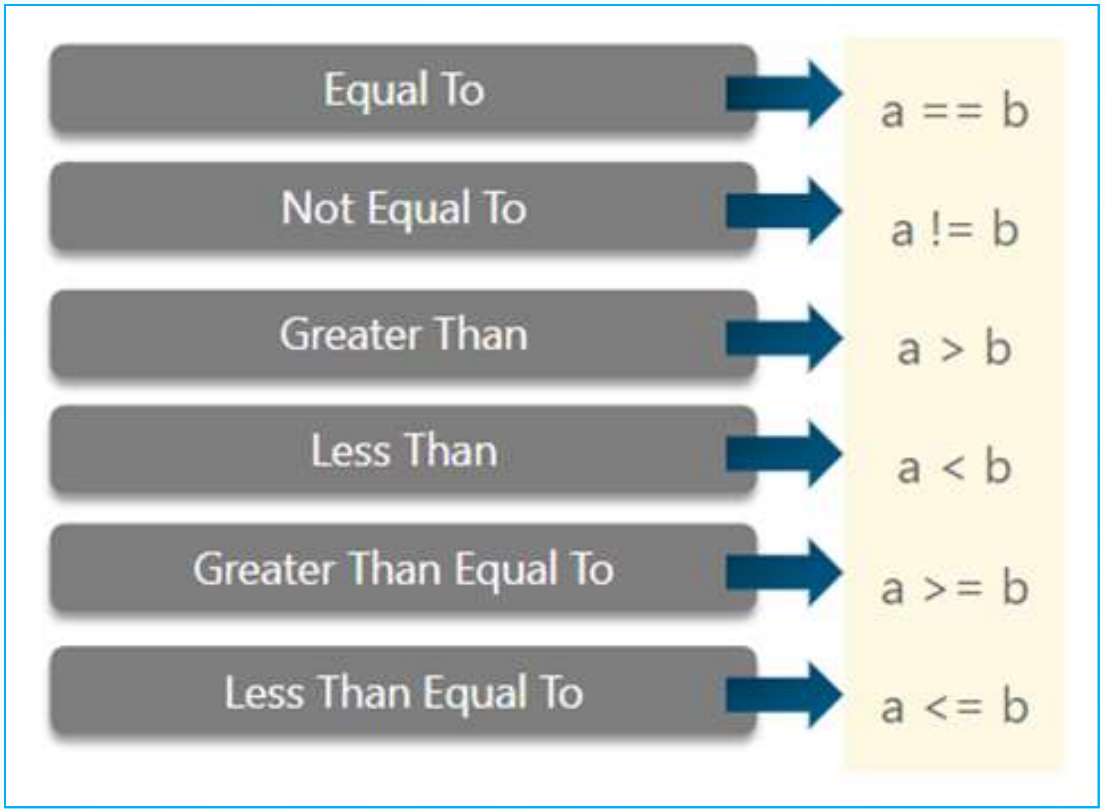
Arithmetic Operators

- **Arithmetic**
- Comparison
- Assignment
- Logical
- Bitwise
- Identity
- Membership



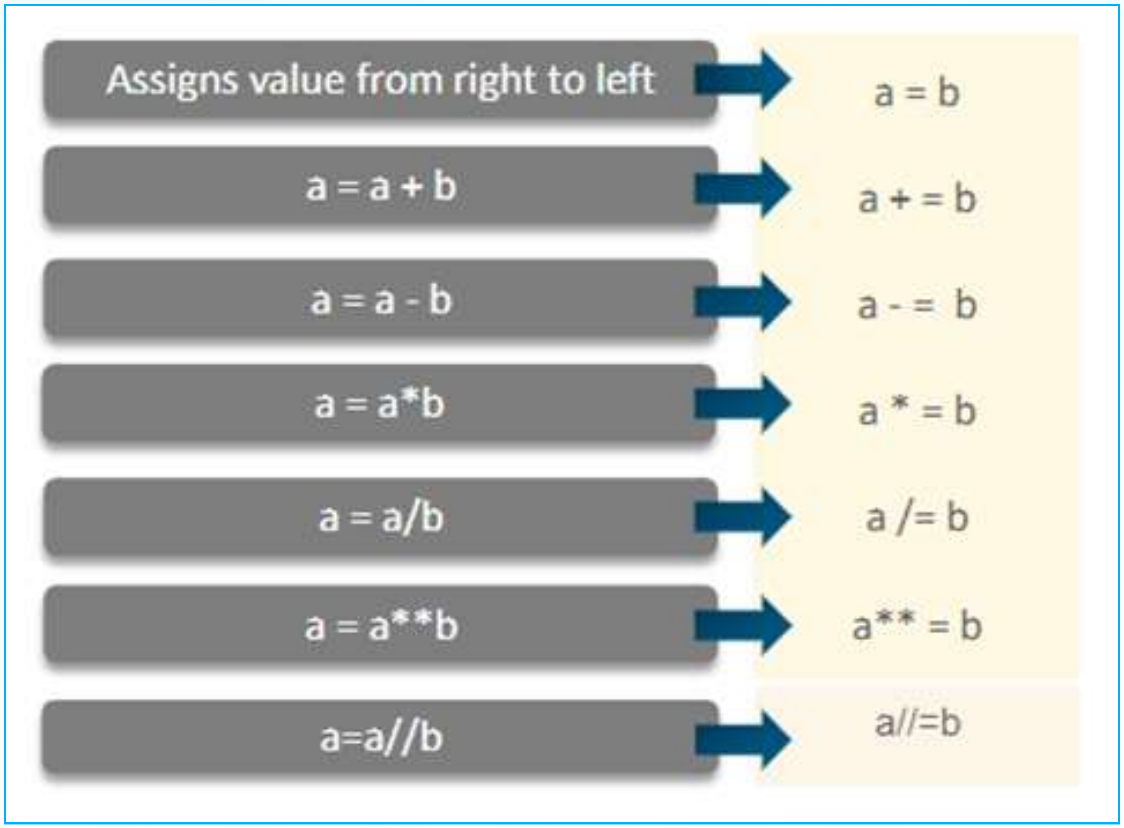
Comparison Operators

- Arithmetic
- **Comparison**
- Assignment
- Logical
- Bitwise
- Identity
- Membership



Assignment Operators

- Arithmetic
- Comparison
- **Assignment**
- Logical
- Bitwise
- Identity
- Membership



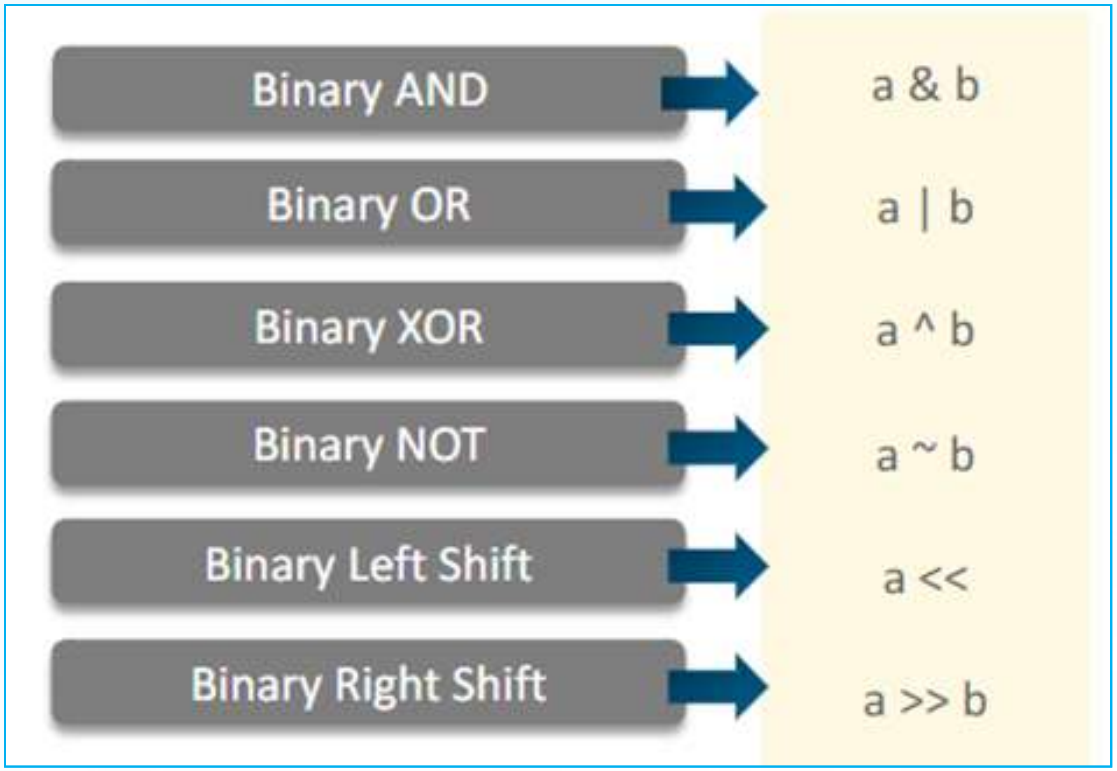
Logical Operators

- Arithmetic
- Comparison
- Assignment
- **Logical**
- Bitwise
- Identity
- Membership

a and b	Returns a, if a is False, b otherwise
a or b	Returns b, if b is False, a otherwise
not a	Returns True, if a is True, False otherwise

Bitwise Operators

- Arithmetic
- Comparison
- Assignment
- Logical
- **Bitwise**
- Identity
- Membership



Identity Operators

- Arithmetic
- Comparison
- Assignment
- Logical
- Bitwise
- **Identity**
- Membership

is	Evaluates to TRUE, if the variables on either side of the operator point to the same object and FALSE otherwise
is not	Evaluates to FALSE, if the variables on either side of the operator point to the same object and TRUE otherwise

Membership Operators

- Arithmetic
- Comparison
- Assignment
- Logical
- Bitwise
- Identity
- **Membership**

in	Evaluates to TRUE, if it finds a variable in the specified sequence and FALSE otherwise
not in	Evaluates to TRUE, if it does not find a variable in the specified sequence and FALSE otherwise

Conditional Statements
&
Loops

if - elif - else

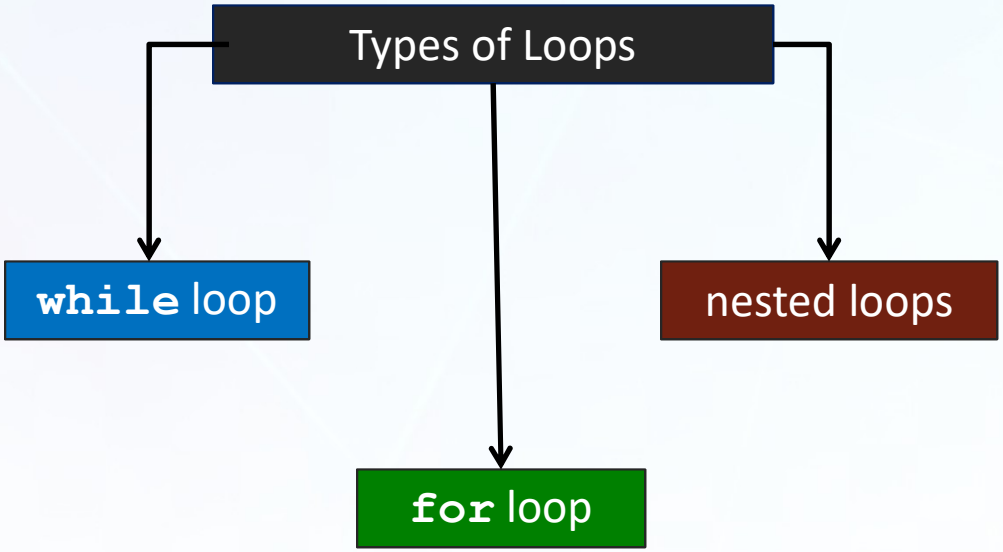
```
age = 15

if(age >= 0 and age <= 12):
    print('Child')
elif(age <= 19):
    print('Teenager')
elif(age <= 35):
    print('Youth')
elif(age <= 50):
    print('Middle aged')
else:
    print('Senior')
```

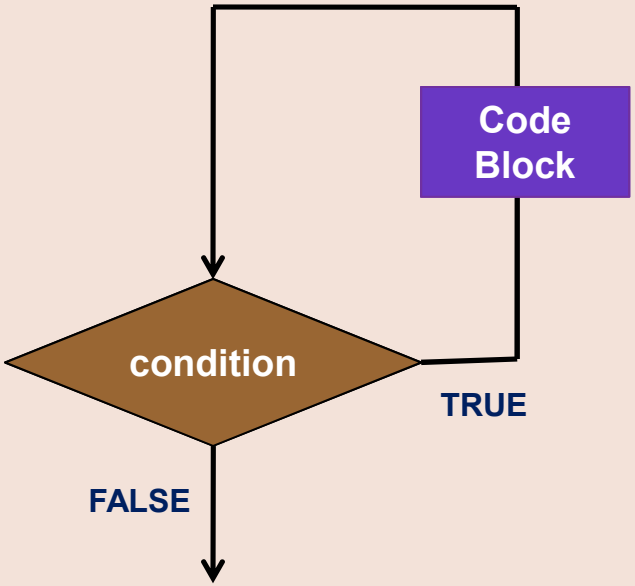
Use : after conditional statement

Follow strict indentation

Loops

A loop statement allows us to execute a block of code multiple times

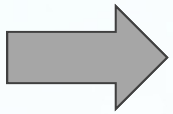




while loop

"While" loop keep iterating until the condition statement evaluates to true. There is no guarantee ahead of time regarding how many times the loop will run.

```
times = 8

while (times > 0):
    print ( "*" * times )
    times -= 1
```



output

```
********
*******
******
*****
****
***
**
*
```

`for` loop

- The **`for`** loop is used to iterate over a sequence (list, tuple, string) or other iterable objects. You can use **`range()`** function to create the sequence.
- A **`for`** loop can have an optional **`else`** block as well. The `else` part is executed if the items in the sequence used in `for` loop exhausts.
- **`break`** statement can be used to stop a `for` loop. In such case, the `else` part is ignored. Hence, a `for` loop's `else` part runs if no `break` occurs.

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
sum = 0

for val in numbers:
    sum = sum + val

print("The sum is", sum)
```

output

```
The sum is 48
```

➢ Iterating over a sequence is called traversal.

for loop

```
for i in range(2, 50, 3):
    if (i > 30):
        break

    print(i, end=' ')
else:
    print("Done!")
```

Use **break** statement to end the loop.

output

```
2 5 8 11 14 17 20 23 26 29
```

```
for i in range(2, 20, 3):
    print(i, end=' ')
else:
    print("Done!")
```

output

```
2 5 8 11 14 17 Done!
```

else: block gets executed after **for** loop exhausts its iterations.

Nested Loops

Nested loops refer to having loops inside other loops

```
count = 0
for i in range(1, 8) :
    print (str(i) * i, end=" >> ")

    for j in range(0, i):
        count += 1

    print (count)
```

Inner loop

Outer loop

```
1 >> 1
22 >> 3
333 >> 6
4444 >> 10
55555 >> 15
666666 >> 21
7777777 >> 28
```

Loop Control Statements

Types of Loops

break	continue	pass
Terminates the loop and transfers execution to statement immediately following the loop.	Causes the loop to skip the reminder of its body and continues to the next hop of the loop	Serves as a place-holder when there is no code to execute but need some code syntactically.

```
for i in range (1, 20) :
    print (i, end = " ")
    if (i == 5) :
        break
print ('\nloop ends')
```

```
1 2 3 4 5
loop ends
```

```
for i in range (1, 10) :

    if (i % 2 == 0) :
        continue
    else:
        print (i, end = " ")
print ('\nloop ends')
```

```
1 3 5 7 9
loop ends
```

```
for k in range(1, 10):
    pass
print ('\nloop ends')
```

```
loop ends
```

Python Comprehensions

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined.

Types of Comprehensions

- **List Comprehensions**
- **Dictionary Comprehensions**
- **Set Comprehensions**
- **Generator Comprehensions**

List comprehension

List Comprehensions provide an elegant way to create new lists.

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]

# Using loop for constructing output list
output_list = []
for var in input_list:
    if var % 2 == 0:
        output_list.append(var)
print("Output List using for loop:", output_list)

# using list comprehension
list_using_comp = [var for var in input_list if var % 2 == 0]
print("Output List using list comprehensions:", list_using_comp)
```

Dictionary comprehension

Dictionary comprehensions provide an elegant way to create new dictionaries, just like lists.

```
input_list = [1, 2, 3, 4, 5, 6, 7]

output_dict = {}

# Using loop for constructing output dictionary
for var in input_list:
    if var % 2 != 0:
        output_dict[var] = var**3

print("Output Dictionary using for loop:", output_dict )

# using dictionary comprehension
dict_using_comp = {var:var ** 3 for var in input_list if var % 2 != 0}

print("Output Dictionary using dictionary comprehensions:", dict_using_comp)
```

Set comprehension

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined.

```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]

set_using_comp = {var for var in input_list if var % 2 == 0}

print("Output Set using set comprehensions:", set_using_comp)
```

Generator Comprehension

- Generator Comprehensions are very similar to list comprehensions.
- One difference between them is that generator comprehensions use circular brackets whereas list comprehensions use square brackets.
- The major difference between them is that generators don't allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient.

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]

# Note that we use () brackets here unlike list [] brackets
output_gen = (var for var in input_list if var % 2 == 0)

print("Output values using generator comprehensions:", end = ' ')

for var in output_gen:
    print(var, end = ' ')
```

Functions

Functions

function is a block of organized and reusable code that performs a specific action

Function Definition

```
def fun(a, b, c):
    print (a, b, c)
    return()
```

The `return` statement terminates the execution and returns control to the calling function

Function Call

```
fun(10, 20, 30)
```

```
10 20 30
```

Variable Scope

Global Variables

Variables that are declared outside a function are global in scope. They can be used anywhere in the program.

Local Variables

Variables that are declared inside a function can be used within the function only.

```
i = 10

def fun():
    j = 20
    print("j = ", j)
    print("i = ", i)

fun()
print("i = ", i)
print("j = ", j)
```

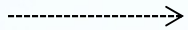
```
j =  20
i =  10
i =  10
Traceback (most recent call last):

  File "<ipython-input-267-8422d559797c>",
line 11, in <module>
    print("j = ", j)

NameError: name 'j' is not defined
```

Because ***j*** is ***local*** to the function it can not be accessed outside the function.

Variable Scope

A global variable's value can not be changed inside a function. Only the value of the variable will be passed to the function, but not its actual reference.

```
def change_name(name):
    name = "Bill"

name = "Gates"
change_name(name)
print(name)
```



Prints "Gates", not "Bill"

Using global

We can get a reference to a global variable inside a function by using "**global**" keyword.

```
def change_name():
    global name
    name = "Bill"

name = "Gates"
change_name()
print(name)
```

Prints "Bill" now

Return multiple values

A function can return multiple values by separating the values by comma

```
def sum_diff(n1, n2):
    return n1+n2, n1-n2

s, d = sum_diff(20, 10)

print("20 + 10 = ", s)
print("20 - 10 = ", d)
```

Function Arguments

Function Arguments

1. Required Arguments
2. Default Arguments
3. Keyword Arguments
4. Variable-length Arguments
5. Variable-length Keyword Arguments

Required Arguments

Required arguments are the arguments passed to a function in correct positional order. The number of arguments in the function call should exactly match with the function definition.

```
def fun(a, b, c):
    print (a, b, c)
    return()

fun(2,3,4)

fun(1,2)
```

```
2 3 4
Traceback (most recent call last):

  File "<ipython-input-268-293138de4ef6>",
line 8, in <module>
    fun(1,2)

TypeError: fun() missing 1 required
positional argument: 'c'
```

An error is thrown because the number of function arguments did not match that of function definition.

Keyword Arguments

When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

```
def fun(a, b, c):
    print("a: %d, b: %d, c: %d" % (a, b, c))
    return()

fun(a=10, c=30, b=20)
fun(b=20, c=30, a=10)
```

```
a: 10, b: 20, c: 30
a: 10, b: 20, c: 30
```

Arguments are passed by name

Default Arguments

A default argument assumes a default value if a value is not provided in the function call for that argument.

```
def fun(a, b, c=100):
    print("a: %d, b: %d, c: %d" % (a, b, c))
    return()

fun(10, 20, 30)
fun(10, 20)
```

Here the default value is used.

```
a: 10, b: 20, c: 30
a: 10, b: 20, c: 100
```

default value is used.

Variable Length Arguments

- Variable length arguments allow us to process more arguments than you specified while defining the function.
- *variable-length* arguments are not named in the function definition.

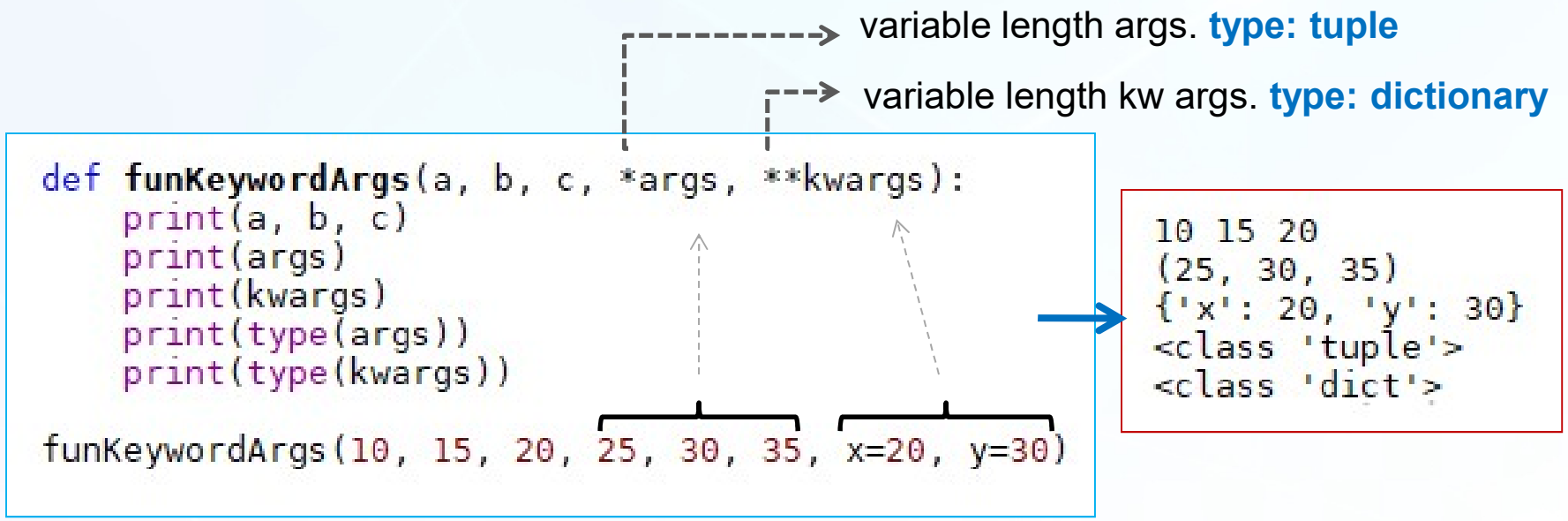
An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments. All additional arguments passed are treated as a tuple.

```
def funArbitrary(a, b, c=20, *args):
    print(a, b, c)
    print(args)
    print(a + b + c + sum(args))

funArbitrary(32, 44, 120, 23, 34, 45)
```

```
32 44 120
(23, 34, 45)
298
```

Variable length arguments

Variable Length Keyword Args



Unpacking Argument Lists

- The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments.
- For instance, the built-in range() function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the * operator to unpack the arguments out of a list or tuple:

```
lst = [3, 7]
print ( range (*lst) )
print (*lst)
```

```
range(3, 7)
3 7
```

Unpacking Argument Lists

- Dictionaries can deliver keyword arguments with the **- operator

```
def myAddress(location, city='Hyderabad', state='Telangana'):
    print("Location: ", location)
    print("City: ", city)
    print("State: ", state)

addr = {"location": "Sundar Akash Apartments", "city": "Pune", "state": "Maharashtra"}
myAddress(**addr)
```

```
Location:  Sundar Akash Apartments
City:  Pune
State:  Maharashtra
```

Working with
Command line Arguments

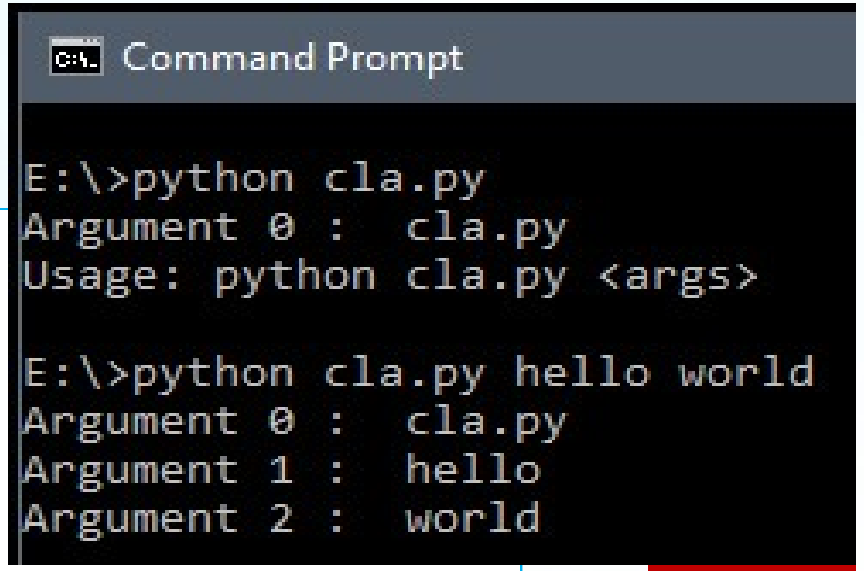
Command-line Arguments

- `sys.argv` provides the list of command-line arguments passed to the Python program.
- `argv` represents all the items that come along via the command-line input. It's basically an array holding the command line arguments of our program.
- The first argument, `sys.argv[0]`, is always the name of the program as it was invoked, and `sys.argv[1]` is the first argument you pass to the program.

Command-line Arguments

code

```
import sys

program_name = sys.argv[0]
arguments = sys.argv[1:]
count = len(arguments)
i = 0

for x in sys.argv:
      print ("Argument", i, ": ", x)
      i += 1

if len (sys.argv) < 2 :
    print ("Usage: python cmd-line-args.py <args>")
    sys.exit (1)
```



output

Exception Handling

Exceptions

- Errors detected during execution are called *exceptions* and are not unconditionally fatal.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

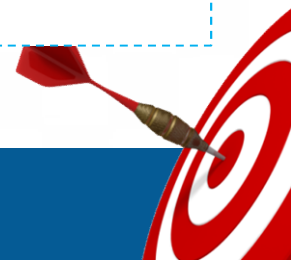
Handling Exceptions

- In Python, we can handle exception using **try** block and catch the exceptions using **except** block

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

try .. except ..

- First, the try clause (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If no handler is found, it is an unhandled exception and execution stops with an exception statement.
- You can catch all unhandled exception with a simple "**except:**" clause with out specifying any exceptions



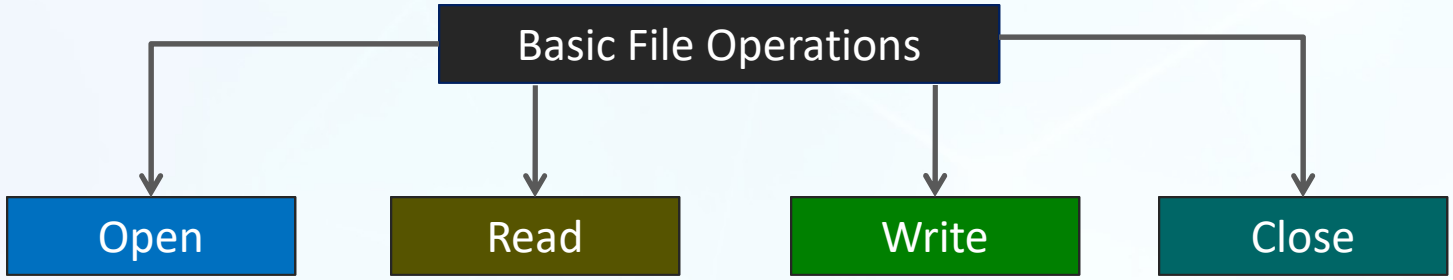
try .. except ..

```
try:
    ... your code here ...

except FileNotFoundError as fne:
    print(fne.strerror)

except (NameError, KeyError) as e:
    pass

except Exception as e:
    print(e.with_traceback)

finally:
    .. this code always gets executed ..
```

Can catch multiple exceptions

Basic File Operations

File Operations




```
f_in = open("anthem1.txt", "r")
f_out = open("anthem1_out.txt", "w")

i = 1
for line in f_in:
    f_out.write(str(i) + ": " + line)
    i = i + 1

f_in.close()
f_out.close()
```

Open the file in read mode

Open the file in write mode

Read from a file opened in read mode

Write to a file opened in write mode

Close the opened files

Python File Access Modes

Mode	Description
'r'	Open a file for reading. **(default)**
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'b'	Open in binary mode.
'+'	Open a file for updating (reading and writing)

Note: ***Modes can be mixed***. Ex: ab+ : opens file for appending and reading in binary mode

Reading a File

- To read a file, we must open the file in read mode.
- Use the **`read(n)`** method to read ***n*** characters of data. **`read()`** method reads and returns up to the end of the file.
- We can change our current file cursor using the **`seek()`** method. Similarly, the **`tell()`** method returns our current position (in number of bytes).

```
fh = open("anthem1.txt")
fh.tell()          ------>  0

fh.read(7)         ------>  'Thou ar'
fh.tell()          ------>  7

fh.read()          ------>  't, the ruler …. till the end of the file
fh.tell()          ------>  504

fh.seek(4)         ------>  Move the cursor location to 4

fh.read(15)        ------>  ' art, the ruler'
```

Write to a file

- To write into a file, we need to open it in write `'w'`, append `'a'` or exclusive creation `'x'` mode.
- Writing a string or sequence of bytes (for binary files) is done using **`write()`** method, which returns the number of characters written to the file.

```
with open("test.txt", 'w', encoding = 'utf-8') as f:
    f.write("my first file\n")
    f.write("This file\n\n")
    f.write("contains three lines\n")
```

---> Using **`with`** notation you don't need to explicitly `close` the file.

Renaming a File

- Use **`rename`** method.

```
f = os.rename("file.txt", "new_file.txt")
```

Deleting a File

- Use `remove` method.

```
os.remove("test.txt")
```

File Object Attributes

- You can get various file attributed from an opened file object

`file.name`	The name of the file
`file.closed`	True/False. True if closed, False otherwise.
`file.mode`	The value of the I/O mode parameter
`file.encoding`	The encoding that this file uses.


THANK
YOU