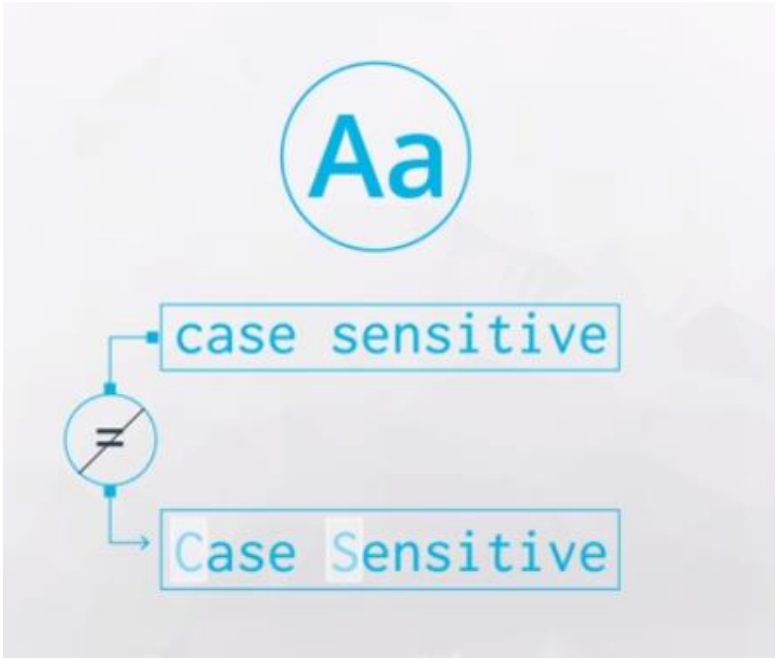
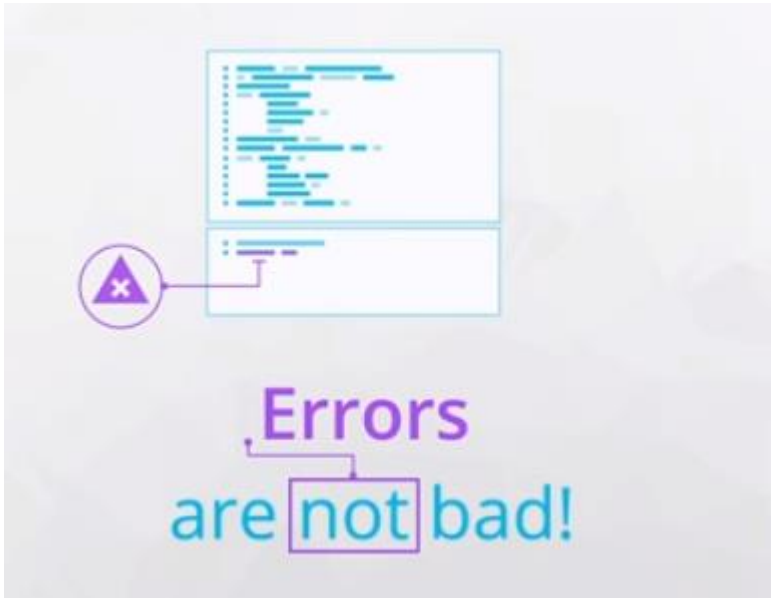
# Python for Analytics


Aa
case sensitive
≠
Case Sensitive


spacing
matters


Errors
are not bad!

# **Data Types -** Integers and Floats

There are two Python data types that could be used for numeric values:

- **int** - for integer values
- **float** - for decimal or floating point values

# Variables & Assignment Operators

```
my_population = 74728
```

1. Only use ordinary letters, numbers and underscores in your variable names. They can't have spaces, and need to start with a letter or underscore.
2. **You can't use reserved words or built-in identifiers**
3. The pythonic way to name variables is to use all lowercase letters and underscores to separate words.

**YES**

```
my_height = 58
my_lat = 40
my_long = 105
```

**NO**

```
my height = 58
MYLONG = 40
MyLat = 105
```

Keywords in Python programming language

| | | | | |
|---|---|---|---|---|
| False | class | finally | is | return |
| None | continue | for | lambda | try |
| True | def | from | nonlocal | while |
| and | del | global | not | with |
| as | elif | if | or | yield |
| assert | else | import | pass | |
| break | except | in | raise | |

# Variables & Assignment Operators

```
mv_population = 74728
mv_population = mv_population + 4000 - 600
```

```
78128
```

# Variables & Assignment Operators

```
mv_population = 74728
mv_population = mv_population + 4000 - 600
```

```
78128
```

```
mv_population = 74728
mv_population += 4000 - 600
```

# Variables & Assignment Operators

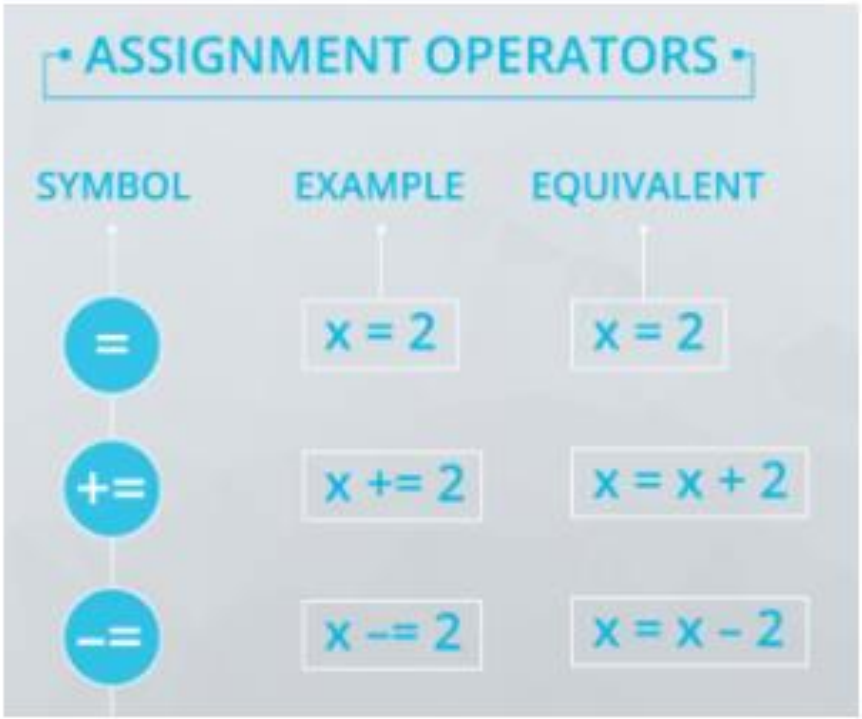
```
mv_population = 74728
mv_population = mv_population + 4000 - 600
```

```
78128
```

```
mv_population = 74728
mv_population += 4000 - 600
```

```
78128
```

# Variables & Assignment Operators



# **Boolean -** data type

# **Boolean -** data type

A BOOLEAN IS A DATA TYPE THAT CAN HAVE A VALUE OF TRUE OR FALSE.

**True = 1**

# **Boolean -** data type

A BOOLEAN IS A DATA TYPE THAT CAN HAVE A VALUE OF TRUE OR FALSE.

**True = 1**

**False = 0**

# **Boolean -** data type

```
x = 42 > 43
print(x)
```

# **Boolean -** data type

```
x = 42 > 43
print(x)
```

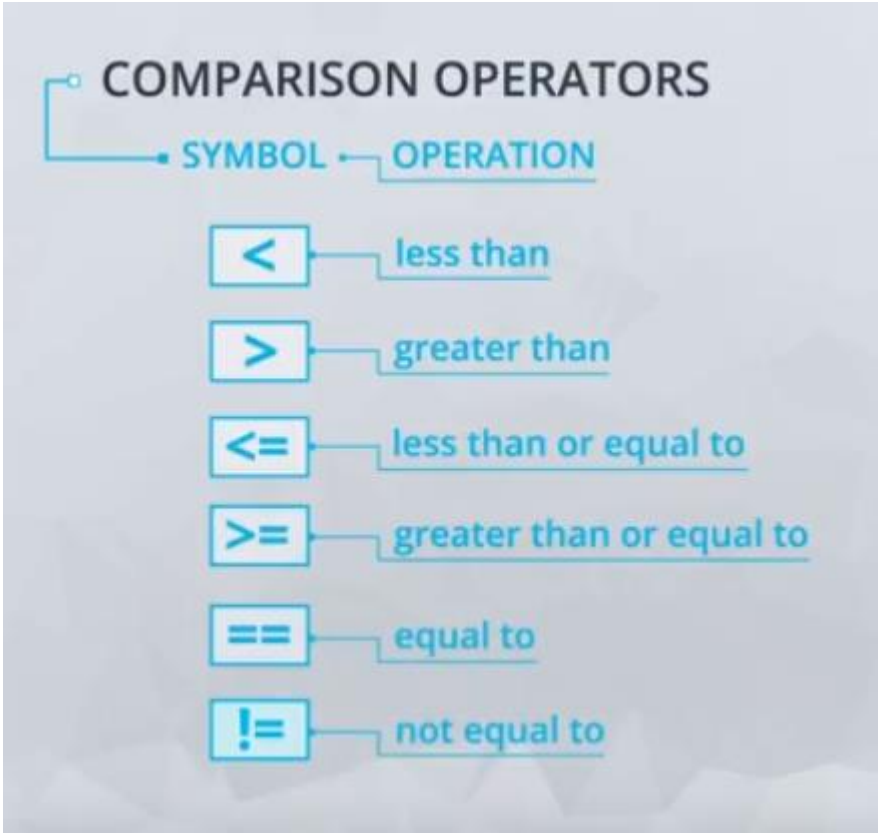
```
False
```

# Comparison Operators

- Boolean result

# Comparison Operators

## - Boolean result



# Comparison Operators

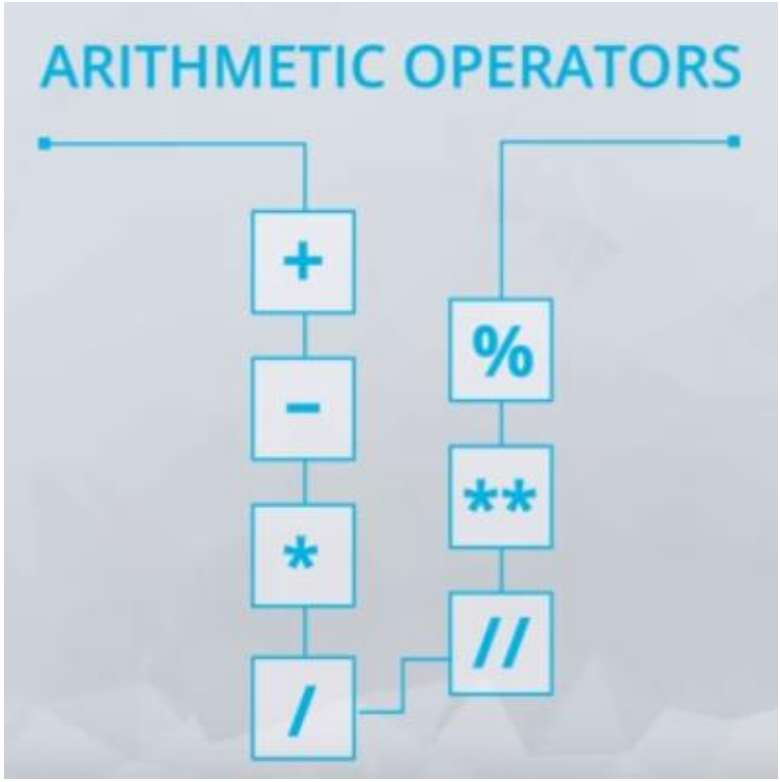
## - Boolean result

Comparison Operators

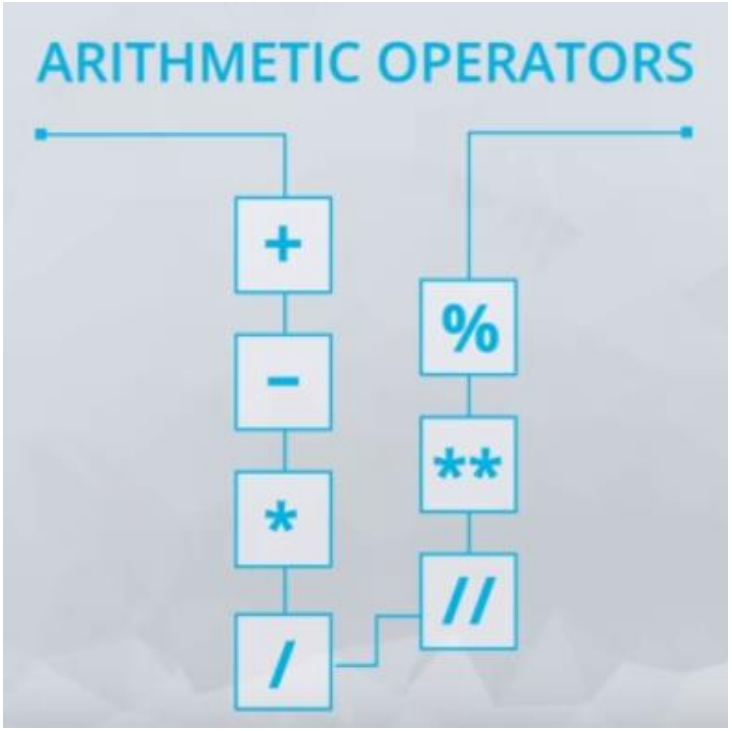
| Symbol Use Case | Bool | Operation |
|---|---|---|
| 5 < 3 | False | Less Than |
| 5 > 3 | True | Greater Than |
| 3 <= 3 | True | Less Than or Equal To |
| 3 >= 5 | False | Greater Than or Equal To |
| 3 == 5 | False | Equal To |
| 3 != 5 | True | Not Equal To |

# Operators

# ARITHMETIC OPERATORS



# ARITHMETIC OPERATORS




## Arithmetic Operators

Arithmetic operators

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `%` Mod (the remainder after dividing)
- `**` Exponentiation (note that `^` does not do this operation, as you might have seen in other languages)
- `//` Divides and rounds down to the nearest integer

# Arithmetic Operators
## - Order of Operation

```
print(1 + 2 + 3 * 3)
```

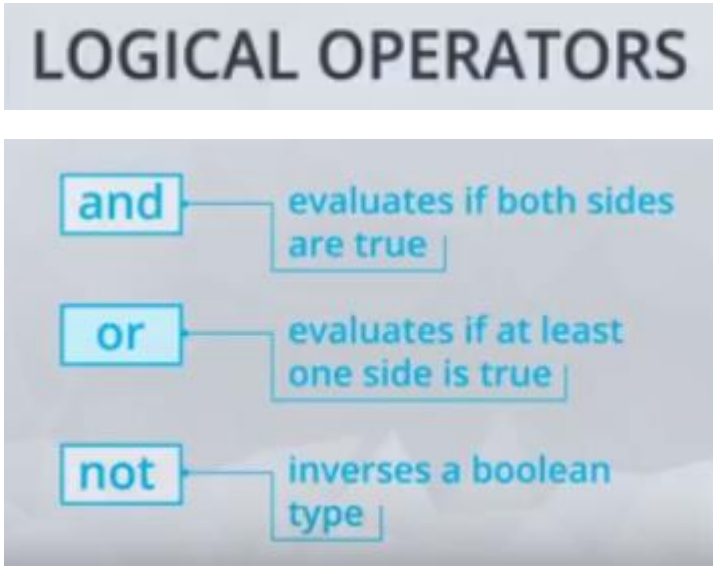
# Arithmetic Operators
# - Order of Operation

**PEMDAS**

1. Parentheses
2. Exponents
3. Multiplication and Division
4. Addition and Subtraction

```
print(1 + 2 + 3 * 3)
```

# Logical Operators



# Boolean Results – Comparison/Logical Operators

| Logical Use | Bool | Operation |
| --- | --- | --- |
| 5 < 3 `and` 5 == 5 | False | `and` - Evaluates if all provided statements are True |
| 5 < 3 `or` 5 == 5 | True | `or` - Evaluates if at least one of many statements is True |
| `not` 5 < 3 | True | `not` - Flips the Bool Value |

# String - Data Type

# String - Data Type

```
print("hello") # double quotes
print('hello') # single quotes
```

```
hello
hello
```

# String - Data Type

```
pet_halibut = “Why should I be tarred
with the epithet “loony” merely
because I have a pet halibut?”
```

# String - Data Type

```
pet_halibut = “Why should I be tarred
with the epithet “loony” merely
because I have a pet halibut?”
```

```
SyntaxError: invalid syntax
```

```
pet_halibut = 'Why should I be tarred
with the epithet "loony" merely
because I have a pet halibut?'
```

# String - Addition / Multiplication

```
first_word = "Hello"
second_word = "There"
print(first_word + second_word)
```

```
word = "Hello"
print(word * 5)
```

# String - Addition / Multiplication

```
first_word = "Hello"
second_word = "There"
print(first_word + second_word)
```

```
HelloThere
```

```
word = "Hello"
print(word * 5)
```

```
HelloHelloHelloHelloHello
```

# String Methods

Methods actually are functions that belong to an object/specific to the data type and are called using **dot notation**.

# String Methods

For example, `lower()` is a string method that can be used like this, on a string called "sample string": `sample_string.lower()`.

# String Methods

some methods that are possible with any string.

```
my_string = "sebastian thrun"

my_string.
```

| | | | | | |
|---|---|---|---|---|---|
| capitalize() | encode() | format() | isalpha() | islower() | istitle() |
| casefold() | endswith() | format_map() | isdecimal() | isnumeric() | isupper() |
| center() | expandtabs() | index() | isdigit() | isprintable() | join() |
| count() | find() | isalnum() | isidentifier() | isspace() | ljust() |

```
>>> my_string.islower()
True
>>> my_string.count('a')
2
>>> my_string.find('a')
3
```

# String Methods

One important string method: `format()`

```
# Example 1
print("Mohammed has {} balloons".format(27))
```

```
Mohammed has 27 balloons
```

# String Methods

```
# Example 2
animal = "dog"
action = "bite"
print("Does your {} {}?".format(animal, action))
```

```
Does your dog bite?
```

# String Methods

## Another important string method: `split()`

1. A basic split method:

```
new_str = "The cow jumped over the moon."
new_str.split()
```

Output is:

```
['The', 'cow', 'jumped', 'over', 'the', 'moon.']
```

# String Methods

## Another important string method: `split()`

2. Here the separator is space, and the maxsplit argument is set to 3.

```
new_str.split(' ', 3)
```

Output is:

```
['The', 'cow', 'jumped', 'over the moon.']
```

# Type and Type Conversion

```
print(type(633))
print(type("633"))
print(type(633.0))
```

```
<class 'int'>
<class 'str'>
<class 'float'>
```

# Type And Type Conversion

```
house_number = 13
street_name = “The Crescent”
town_name = “Belmont”
print(type(house_number))

address = str(house_number) + “ ” + street_name + “, ” + town_name
print(address)
```

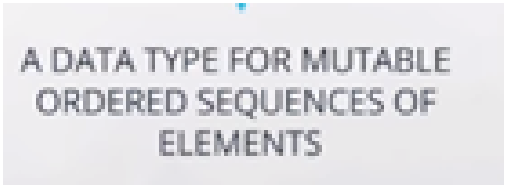
```
<class ‘int’>
13 The Crescent, Belmont
```

# Data structures - List

**Data structures** are containers that organize and group data types together in different ways.

**List** is one of the most common and basic data structures in Python.

# List



# List

A DATA TYPE FOR MUTABLE ORDERED SEQUENCES OF ELEMENTS

```
months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']
```

# List - Indexing

```
months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']
```

```
print(months[0])
print(months[1])
print(months[7])
```

```
January
February
August
```

# Slice and Dice with Lists

```
months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']

q3 = months[6:9]
print(q3)
```

```
['July', 'August', 'September']
```

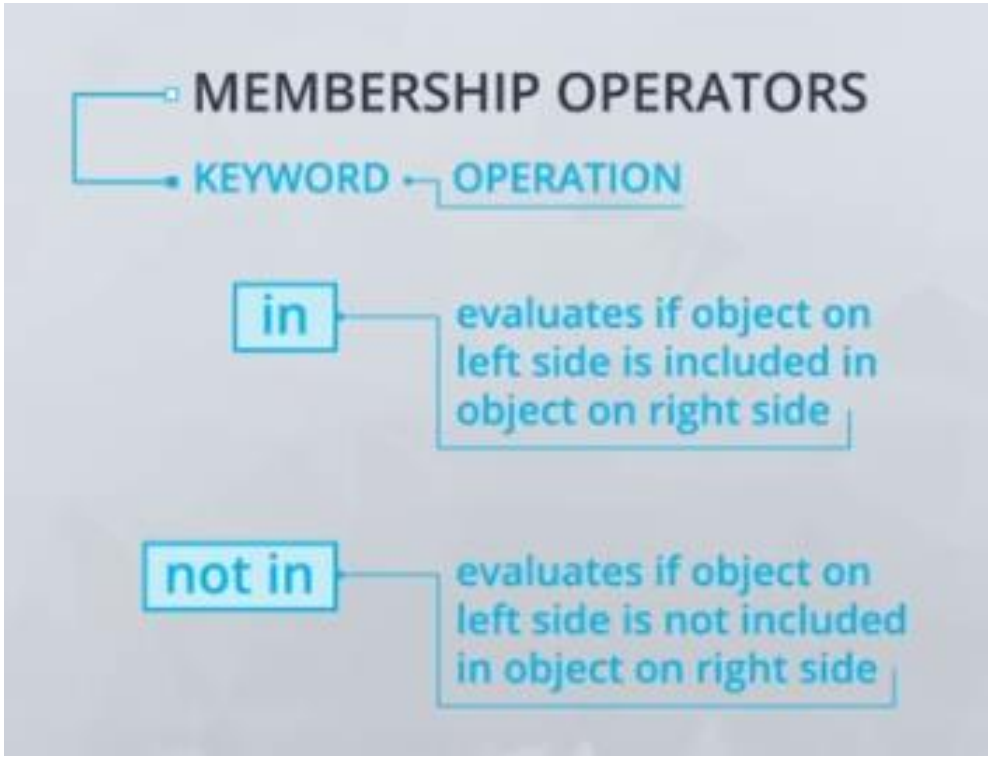
# Slice and Dice with Lists

When using slicing, it is important to remember that the `lower` index is `inclusive` and the `upper` index is `exclusive`.

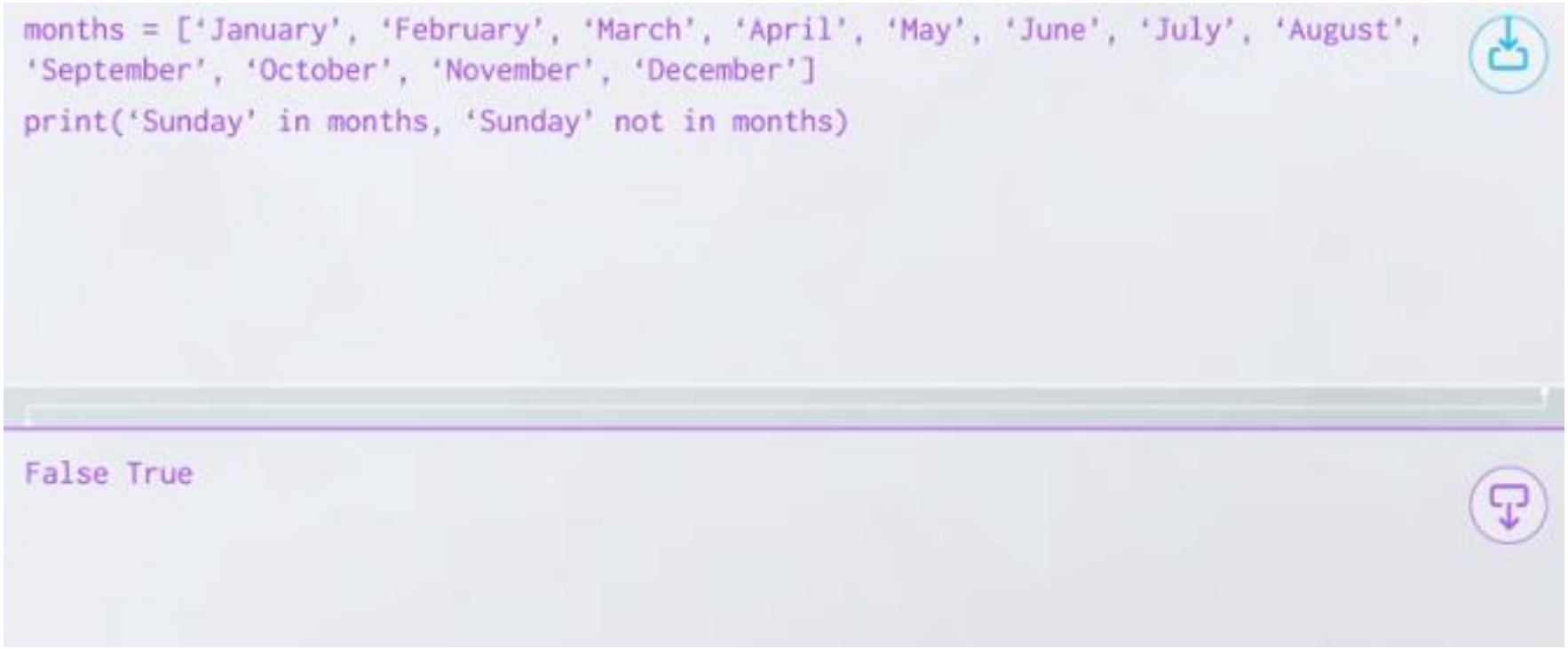
Therefore, this:

```
>>> list_of_random_things = [1, 3.4, 'a string', True]
>>> list_of_random_things[1:2]
[3.4]
```

# Membership Operators



# List and Membership Operators



# Mutability

**1. Mutability** is about whether or not we can change an object once it has been created.

2. If an object can be changed, then it is called **mutable**.

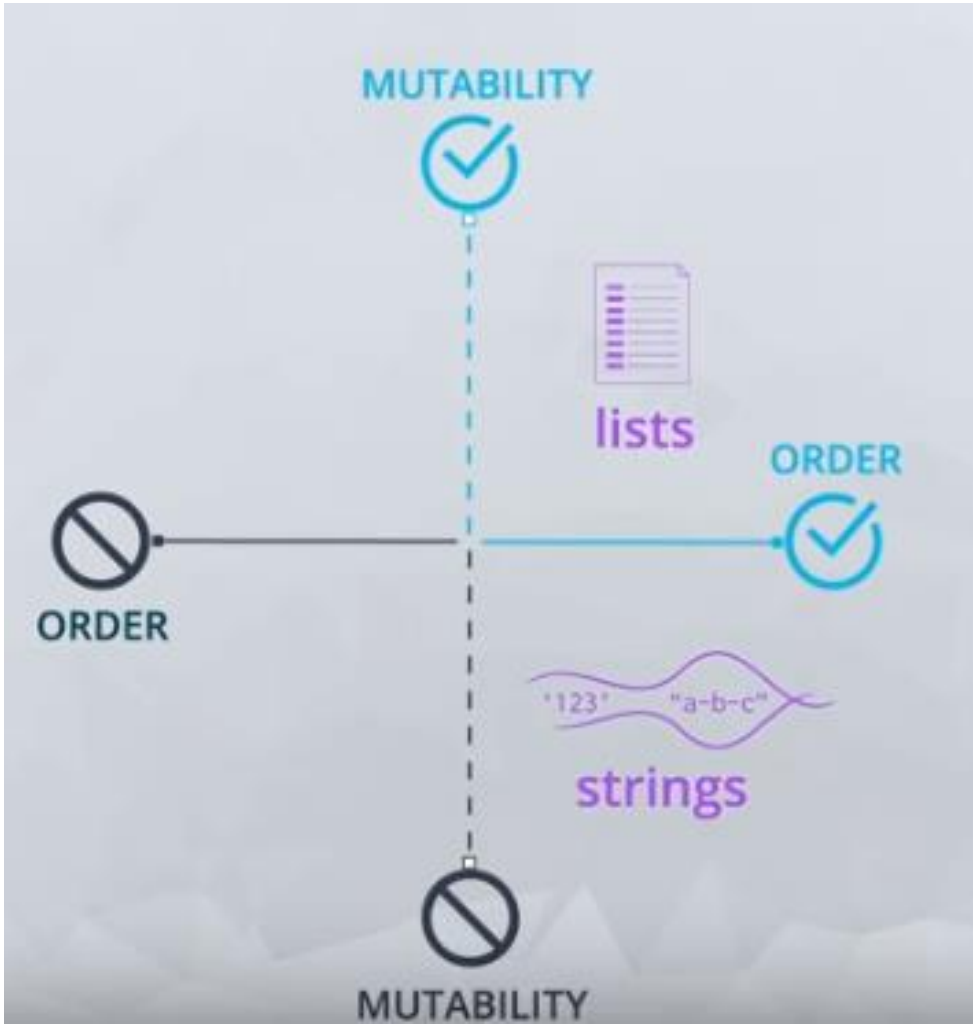
3. If an object cannot be changed without creating a completely new object then the object is considered **immutable**.

# Mutability

```
months = ['January', 'February', 'March', 'April', 'May', 'June','July', 'August',
'September', 'October', 'November', 'December']
months[3] = 'Friday'
print(months)
```

```
['January', 'February', 'March', 'Friday', 'May', 'June','July', 'August',
'September', 'October', 'November', 'December']
```

# Mutability Vs Order



# List Methods

## Join method

Join is a string method that takes a list of strings as an argument, and returns a string consisting of the list elements joined by a separator string.

```
name = "-".join(["García", "O'Kelly"])
print(name)
```

Output:

```
García-O'Kelly
```

# List Methods

## append method

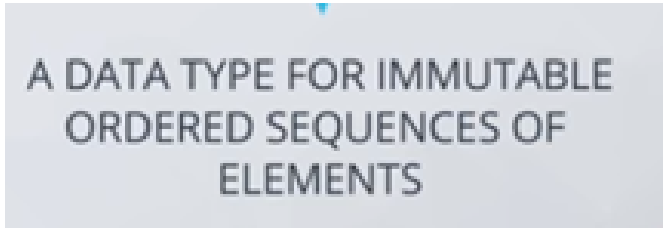
A helpful method called `append` adds an element to the end of a list.

```
letters = ['a', 'b', 'c', 'd']
letters.append('z')
print(letters)
```

Output:

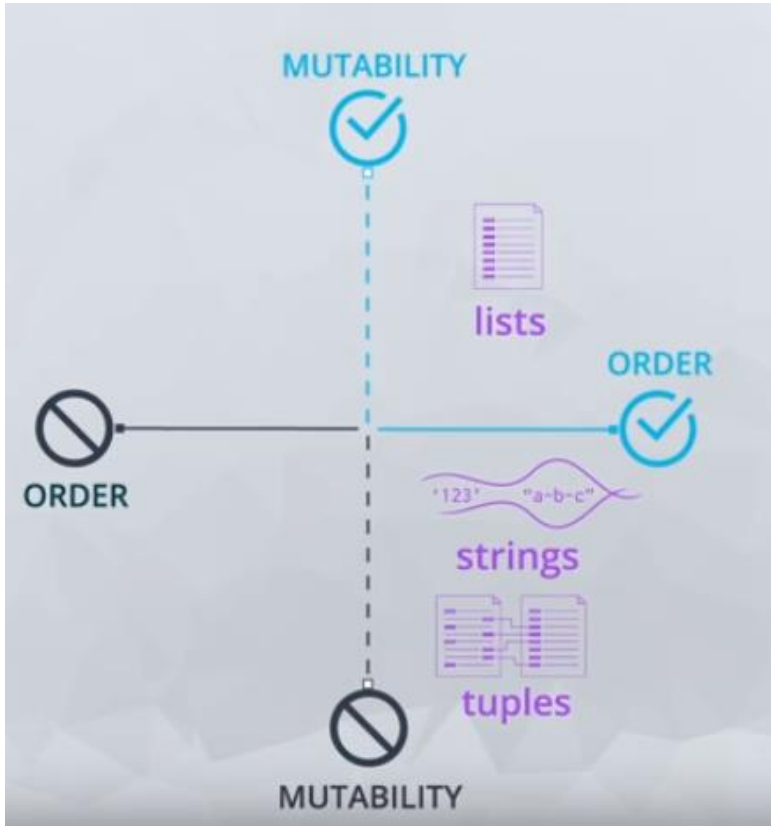
```
['a', 'b', 'c', 'd', 'z']
```

# Tuples



# Tuples

A DATA TYPE FOR IMMUTABLE ORDERED SEQUENCES OF ELEMENTS



# Tuples - Indexing

```
AngkorWat = (13.4125, 103.866667)

print(type(AngkorWat))

print("Angkor Wat is at latitude: {}".format(AngkorWat[0]))
print("Angkor Wat is at longitude: {}".format(AngkorWat[1]))
```

```
<class 'tuple'>
Angkor Wat is at latitude: 13.4125
Angkor Wat is at longitude: 103.866667
```

# SET

A DATA TYPE FOR MUTABLE UNORDERED COLLECTIONS OF UNIQUE ELEMENTS

# SET

```
countries = ['Angola', 'Maldives', 'India', 'United States', 'India', 'Denmark',
'Sweden', 'Ghana', … 777 more countries not displayed]

country_set = set(countries)
print(len(country_set))
```

```
196
```

# SET & Membership operators

```
print('India' in countries)
print('India' in country_set)
```

```
True
True
```

# SET Methods

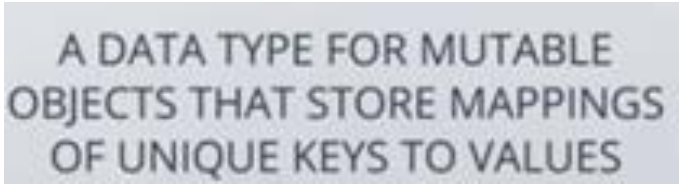
**add method**

```
country_set.add("Italy")
```

# SET Methods

Methods like union, intersection, and difference are easy to perform with sets, and are much faster than such operators with other containers

# Dictionaries




A **dictionary** is a mutable data type that stores mappings of unique keys to values

# Dictionaries Indexing(Key)

```
elements = {'hydrogen': 1,
'helium': 2, 'carbon': 6}

print(elements['carbon'])
```

```
6
```

# Dictionaries - Adding Key/Values

```
elements = {'hydrogen': 1,
'helium': 2, 'carbon': 6}

elements['lithium'] = 3
print(elements)
```

```
{'hydrogen': 1, 'helium': 2,
'carbon': 6, 'lithium':3}
```

# Dictionaries and Membership Operators

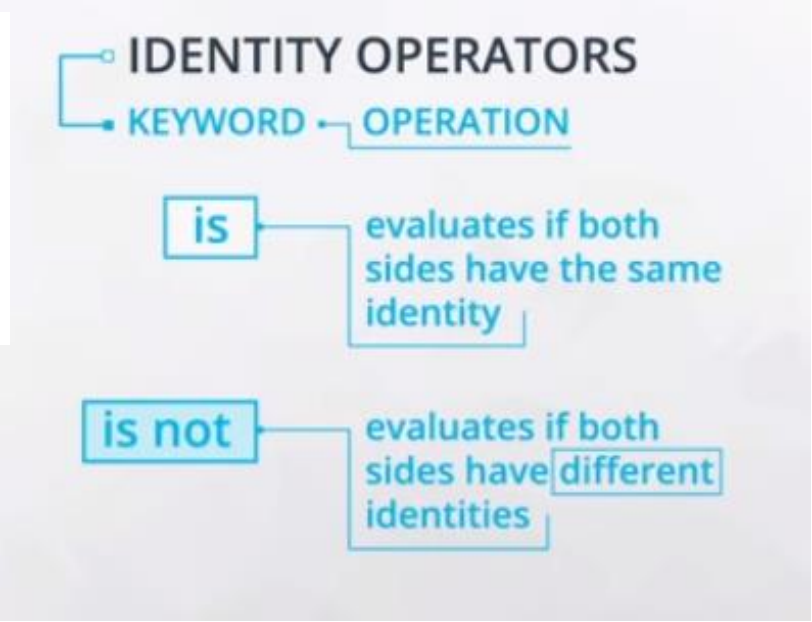
```
elements = {'hydrogen': 1,
'helium': 2, 'carbon': 6}

print('mithril' in elements)
```

```
False
```

# Identity Operators

Identity Operators

| Keyword | Operator |
| --- | --- |
| `is` | evaluates if both sides have the same identity |
| `is not` | evaluates if both sides have different identities |



# Compound Data Structures

We can include containers in other containers to create compound data structures.

```
elements = {'hydrogen': {'number': 1,
                         'weight': 1.00794,
                         'symbol': 'H'},
            'helium': {'number': 2,
                       'weight': 4.002602,
                       'symbol': 'He'}}
```

```
print(elements['helium'])
```

```
{'number': 2, 'symbol': 'He', 'weight':
4.002602}
```

# Compound Data Structures

```
elements = {'hydrogen': {'number': 1,
                         'weight': 1.00794,
                         'symbol': 'H'},
            'helium': {'number': 2,
                       'weight': 4.002602,
                       'symbol': 'He'}}
```

```
print(elements['helium']['weight'])
```

```
4.002602
```

| Data Structure | Ordered | Mutable | Constructor | Example |
|---|---|---|---|---|
| list | Yes | Yes | `[ ]` or `list()` | [5, 'yes', 5.7] |
| tuple | Yes | No | `( )` or `tuple()` | (5, 'yes', 5.7) |
| set | No | Yes | `{ }` or `set()` | {5, 'yes', 5.7} |
| dictionary | No | Keys: No | `{ }` or `dict()` | {'Jun':75, 'Jul':89} |

# Control Flow

- Conditional Statements
- Boolean Expressions
- For and While Loops
- Break and Continue
- Zip and Enumerate
- List Comprehensions

# Conditional Statement - If Statement

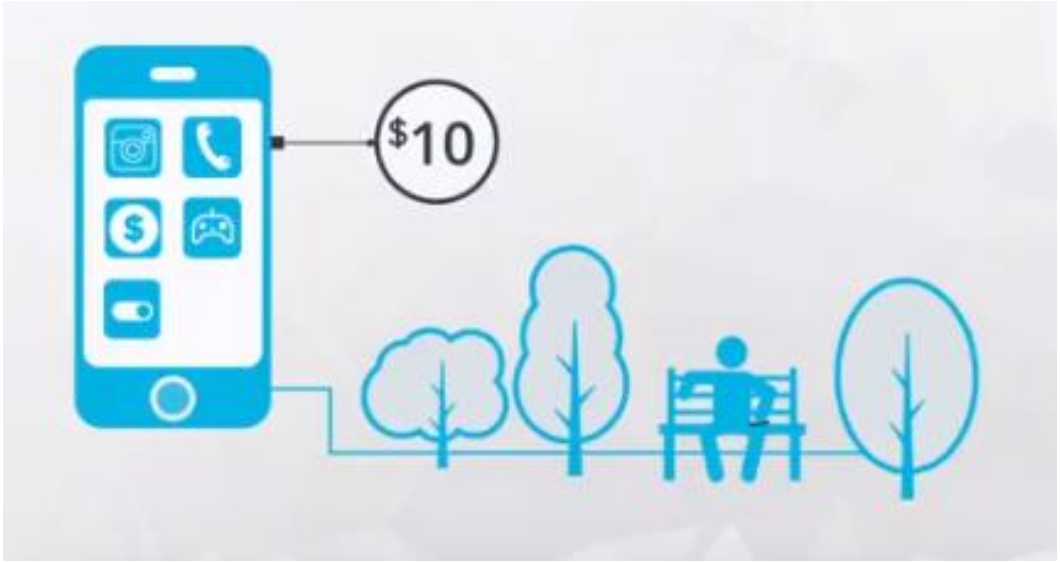
# Conditional Statement - If Statement

## If Statement

An if statement is a conditional statement that runs or skips code based on whether a condition is true or false.

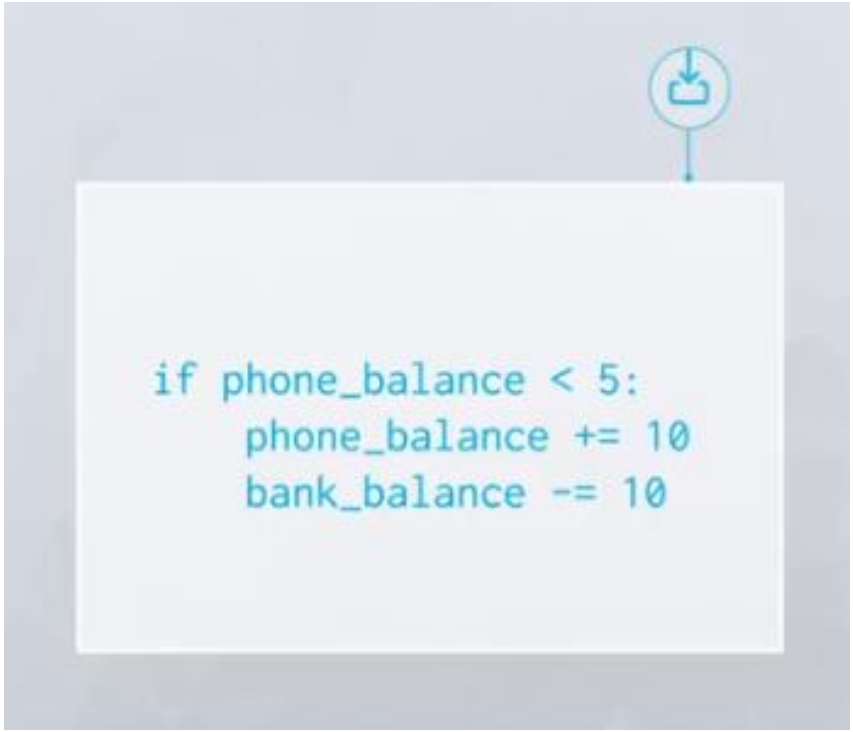
# Conditional Statement - If Statement

# Conditional Statement - If Statement



# Conditional Statement - If Statement

# Conditional Statement - If Statement

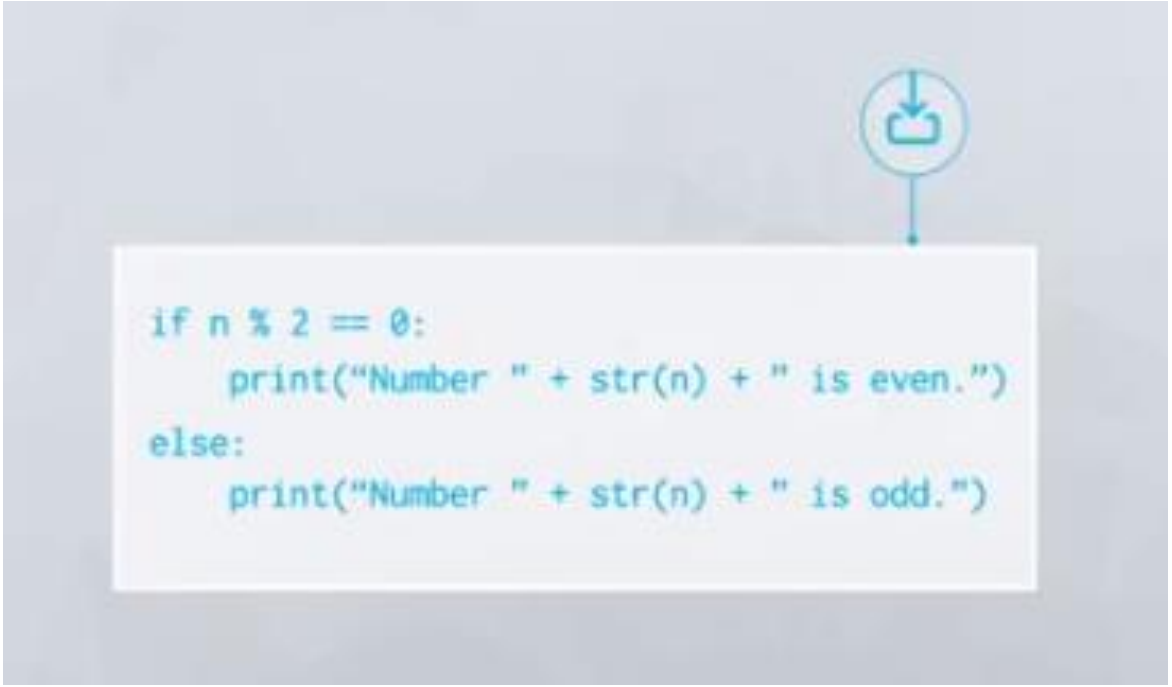


# Conditional Statement - If Statement

```
phone_balance = 3
bank_balance = 100

print(phone_balance, bank_balance)

if phone_balance < 5:
    phone_balance += 10
    bank_balance -= 10

print(phone_balance, bank_balance)
```

```
3, 100
13, 90
```

# Conditional Statement - If else Statement



# Conditional Statement - If else Statement

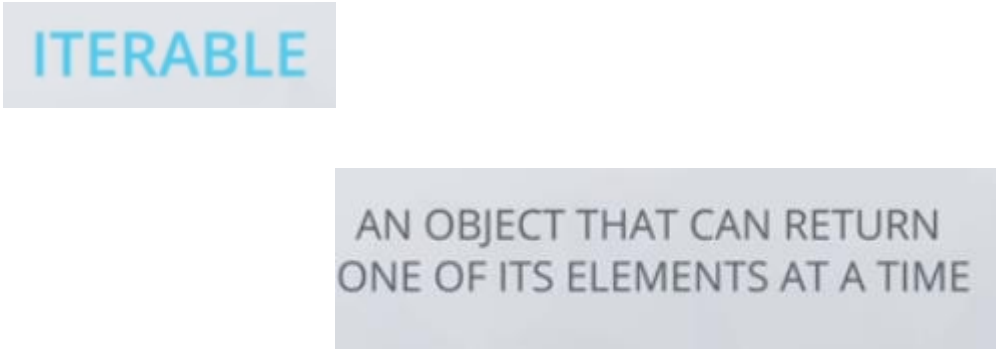
```
n = 4

if n % 2 == 0:
    print("Number " + str(n) + " is even.")
else:
    print("Number " + str(n) + " is odd.")

print(n)
```

```
Number 4 is even.
```

# Loops

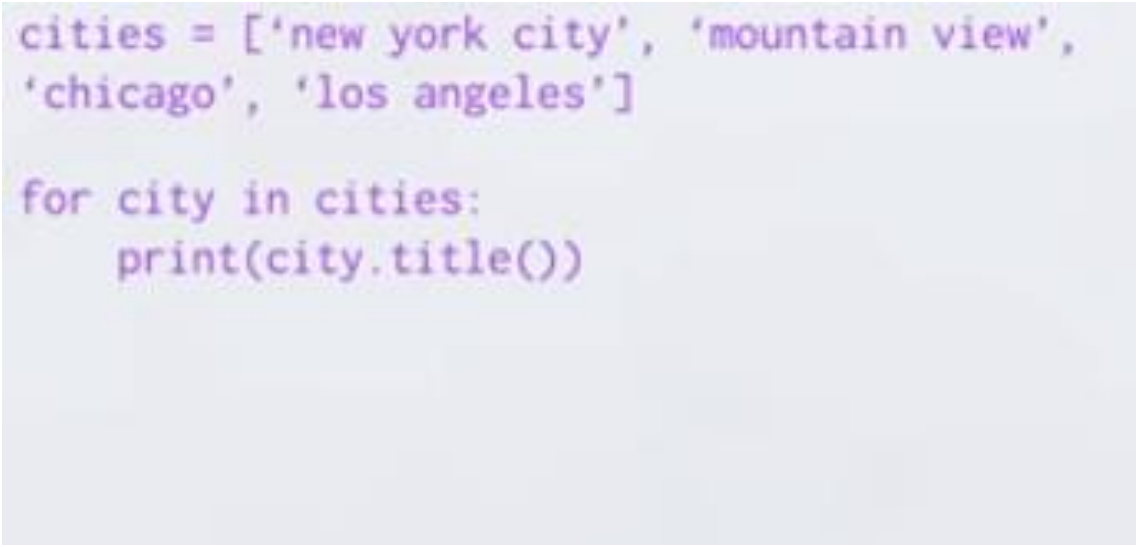
# For Loop

For loop is used to iterate or do something repeatedly, over an **iterable**.



# For Loop

For loop is used to iterate or do something repeatedly, over an **iterable**.



```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']

for city in cities:
    print(city.title())
```

# For Loop

For loop is used to iterate or do something repeatedly, over an **iterable**.

```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']

for city in cities:
    print(city.title())
```

```
New York City
Mountain View
Chicago
Los Angeles
```

# For Loops to modify a list

```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']


for index in range(len(cities)):
    cities[index] = cities[index].title()
```

# For Loops to modify a list

# For Loops to modify a list

```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']


for index in range(len(cities)):
    cities[index] = cities[index].title()
    print(cities)
```

```
['New York City', 'Mountain View', 'Chicago',
'Los Angeles']
```

# For Loops - Iterating Dictionaries

```
cast = {
            "Jerry Seinfeld": "Jerry Seinfeld",
            "Julia Louis-Dreyfus": "Elaine Benes",
            "Jason Alexander": "George Costanza",
            "Michael Richards": "Cosmo Kramer"
        }
```

# For Loops - Iterating Dictionaries

```
cast = {
           "Jerry Seinfeld": "Jerry Seinfeld",
           "Julia Louis-Dreyfus": "Elaine Benes",
           "Jason Alexander": "George Costanza",
           "Michael Richards": "Cosmo Kramer"
       }
```

```
for key in cast:
    print(key)
```

This outputs:

```
Jerry Seinfeld
Julia Louis-Dreyfus
Jason Alexander
Michael Richards
```

# Iterate through both keys and values

```
for key, value in cast.items():
    print("Actor: {}    Role: {}".format(key, value))
```

This outputs:

```
Actor: Jerry Seinfeld    Role: Jerry Seinfeld
Actor: Julia Louis-Dreyfus    Role: Elaine Benes
Actor: Jason Alexander    Role: George Costanza
Actor: Michael Richards    Role: Cosmo Kramer
```

# While Loops

# While Loops

```
card_deck = [4, 11, 8, 5, 13, 2, 8, 10]
hand = []

while sum(hand) <= 17:
    hand.append(card_deck.pop())

print(hand)
```

```
[10, 8]
```

# Break and Continue

BREAK

TERMINATES A FOR
OR WHILE LOOP

```
manifest = [("bananas", 15), ("mattresses", 34),
("dog kennels",42), ("machine", 120),
("cheeses", 5)]
```

```
manifest = [("bananas", 15), ("mattresses", 34),
("dog kennels",42), ("machine", 120),
("cheeses", 5)]

weight = 0
items = []
for cargo in manifest:
        if weight >= 100:
            break
        else:
            items.append(cargo[0])
            weight += cargo[1]
```

```
weight = 0
items = []
for cargo in manifest:
        if weight >= 100:
            break
        else:
            items.append(cargo[0])
            weight += cargo[1]

print(weight)
print(items)
```

```
211
['banana', 'mattresses', 'dog kennels',
'machine']
```

# CONTINUE

TERMINATES ONE ITERATION OF A FOR OR WHILE LOOP

```
fruit = ["orange", "strawberry", "apple"]
foods = ["apple", "apple", "hummus", "toast"]
```

```
fruit_count = 0
for food in foods:
    if food not in fruit:
        print("Not a fruit")
        continue
    fruit_count += 1
    print("Found a fruit!")

print("Total fruit: ", fruit_count)
```

```
Found a fruit!
Found a fruit!
Not a fruit
Not a fruit
Total fruit: 2
```

# Zip and Enumerate

RETURNS AN ITERATOR THAT COMBINES MULTIPLE ITERABLES INTO ONE SEQUENCE OF TUPLES. EACH TUPLE CONTAINS THE ELEMENTS IN THAT POSITION FROM ALL THE ITERABLES.

```
items = ['bananas', 'mattresses', 'dog kennels',
'machine', 'cheeses']

weights = [15, 34, 42, 120, 5]
```

```
for cargo in zip(items, weights):
    print(cargo[0], cargo[1])
```

```
[("bananas", 15), ("mattresses", 34),
("dog kennels",42), ("machine", 120),
("cheeses", 5)]
```

# Unzipping Using (*)

```
manifest = [("bananas", 15), ("mattresses", 34),
("dog kennels",42), ("machine", 120),
("cheeses", 5)]

items, weights = zip(*manifest)

print(items)
print(weights)
```

```
('bananas', 'mattresses', 'dog kennels',
'machine', 'cheeses')
(15, 34, 42, 120, 5)
```

# Enumerate

`enumerate` is a built in function that returns an iterator of tuples containing indices and values of a list. You'll often use this when you want the index along with each element of an iterable in a loop.

```
items = ['bananas', 'mattresses', 'dog kennels',
'machine', 'cheeses']

for i, item in enumerate(items):
    print(i, item)
```

```
0 bananas
1 mattresses
2 dog kennels
3 machine
4 cheeses
```

# List Comprehensions

```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']
```

```
capitalized_cities = []
for city in cities:
    capitalized_cities.append(city.title())
```

```
capitalized_cities = [city.title() for city in cities]
```

```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']

capitalized_cities = [city.title() for city
in cities]
```

```
cities = ['new york city', 'mountain view',
'chicago', 'los angeles']

capitalized_cities = []
for city in cities:
    capitalized_cities.append(city.title())

print(capitalized_cities)
```

# Conditionals in List Comprehensions

```
squares = [x**2 for x in range(9) if x % 2 == 0]
```

```
squares = [x**2 if x % 2 == 0 else x + 3 for x in range(9)]
```

# Functions

```
def population_density(population, land_area):
    """Calculate the population density of an area. """
    return population / land_area
```

# Defining Functions

```
def cylinder_volume(height, radius):
    pi = 3.14159
    return height * pi * radius ** 2
```

```
cylinder_volume(10, 3)
```

# Print

vs

# Return

# Lambda Expressions

```
def multiply(x, y):
    return x * y
```

```
multiply = lambda x, y: x * y
```