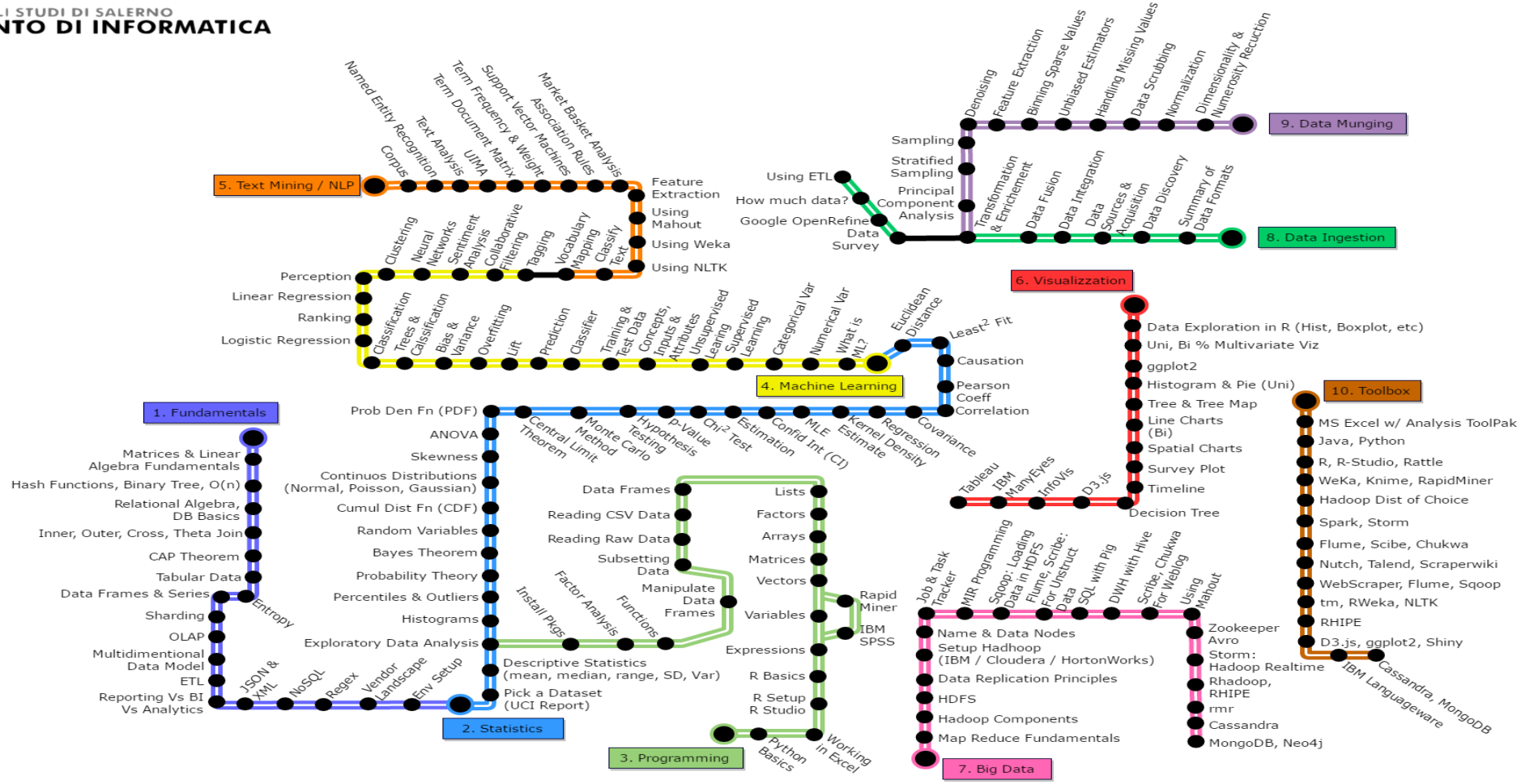
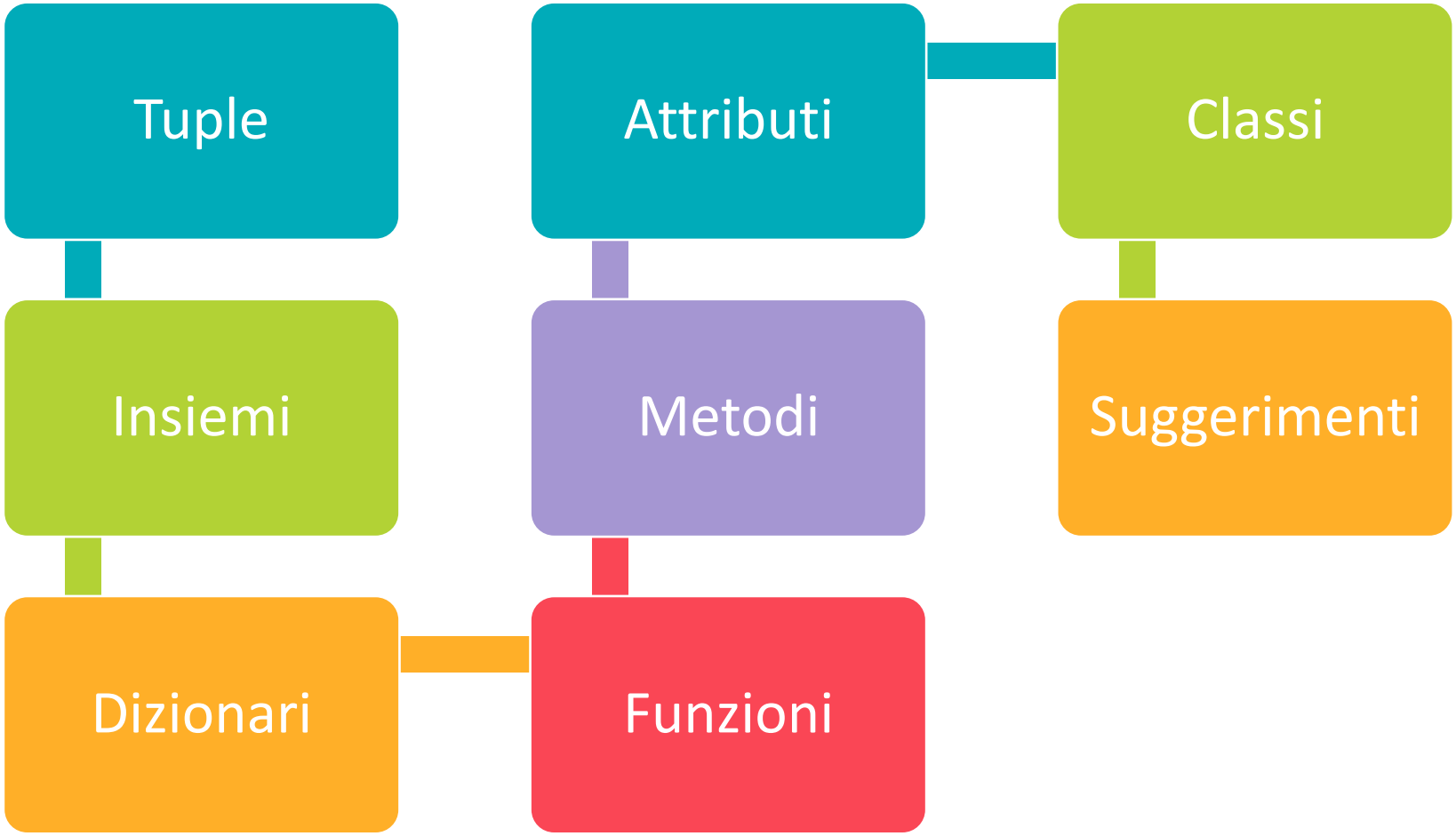


Fondamenti di Data Science e Machine Learning

Introduction to Python 2

Prof. Giuseppe Polese, a.a. 2024-25

Outline


Tuple
Attributi
Classi
Insiemi
Metodi
Suggerimenti
Dizionari
Funzioni

Tuple

Tupla

- Una **tupla** in Python è una **sequenza** o **serie** di **oggetti** Python immutabili molto simili alle liste
- Differenze tra **liste** e **tuple**
  - *A differenza delle liste, gli oggetti delle tuple non possono essere modificati*
  - *Le tuple vengono definite utilizzando le parentesi*

```
# Creating a tuple
tuple_1 = ()
tuple_2 = (1,)
tuple_3 =('a','b','c','d',1,2,3)
```

Tupla: accesso ed eliminazione

```
# Accessing items in tuple
print("3rd item of Tuple: ", tuple_3[2])
print("First 2 items of Tuple: ", tuple_3[0:2])
```

---- output ----
3d item of Tuple: c
First 2 items of Tuple ('a', 'b')

```
# Deleting tuple
print("Same tuple : ", tuple_3)
del tuple_3
print(tuple_3) # Will throw an error message
```

Tupla: operazioni di base (1)

```
# Example code
tuple =('a','b','c','d',1,2,3)
print("Length of Tuple: ", len(tuple))
tuple_concat = tuple + (7,8,9)
print("Concatenate tuples: ", tuple_concat)
print("Repetition: ", (1,'a',2,'b') * 3)
print("Membership check: ", 3 in (1,2,3))
print("Iteration: ")
for x in (1,2,3): print(x)
```

```
---- output ----

 Length of Tuple:  7
Concatinated Tuple:  ('a', 'b', 'c', 'd', 1, 2, 3, 7, 8, 9)
Repetition:  (1, 'a', 2, 'b', 1, 'a', 2, 'b', 1, 'a', 2, 'b')
Membership check:  True
1
2
3
```

Tupla: operazioni di base (2)

```
# Negative sign will retrieve item from right
print("slicing: ", tuple_concat[-2])
print("slicing range: ", tuple_concat[2:])
# Max and Min
print("Max of the tuple:", max((1,2,3,4,5,6,7,8,9,10)))
print("Min of the tuple:", min((1,2,3,4,5,6,7,8,9,10)))
```

```
-------- output ---------
slicing:  8
slicing range:  ('c', 'd', 1, 2, 3, 7, 8, 9)
Max of the tuple: 10
Min of the tuple: 1
```

Insiemi (set)

Set

- Gli **insiemi (set)** sono le implementazioni degli insiemi matematici
- Tre caratteristiche chiave del set sono le seguenti
  - *La raccolta degli articoli **non** è **ordinata***
  - *Nessun articolo duplicato verrà archiviato: ogni **item** è **unico***
  - *I set sono **mutabili**: i suoi elementi possono essere modificati*

```
# Creating a set
languages = set()
print(type(languages), languages)
languages ={'Python','R','SAS','Julia'}
print(type(languages), languages)
# set of mixed datatypes
mixed_set={'Python',(2.7,3.4)}
print(type(mixed_set), mixed_set)
```

```
-------- output --------
<class 'set'> set()
<class 'set'> {'Python', 'R', 'Julia', 'SAS'}
<class 'set'> {'Python', (2.7, 3.4)}
```

Set: accesso ed eliminazione

```
# Accessing set elements
print(type(languages), languages)
print(list(languages)[0])
print(list(languages)[0:3])

print(type(languages), languages)
# add an element
languages.add('C')
print(languages)
languages.update(['Java','SPSS']) # add multiple elements
print(languages)
# add list and set
languages.update(['Ruby','C++'],{'Data Science','AI'})
print(languages)
```

```
-------- output ---------
<class 'set'> {'Python', 'Julia', 'R', 'SAS'}
{'Julia', 'Python', 'R', 'SAS', 'C'}
{'Julia', 'SPSS', 'Python', 'R', 'SAS', 'Java', 'C'}
{'Julia', 'Ruby', 'SPSS', 'Python', 'Data Science', 'C++', 'R', 'SAS', 'AI', 'Java', 'C'}
```

Rimozione di elementi dal Set

```
print(type(languages), languages)
# remove an element
languages.remove('AI')
print(languages)

""" as opposed to remove, discard will not throw an error when discarding the already removed element AI """
languages.discard('AI')
print(languages)

# pop will remove a random item from set
print('Removed:', (languages.pop()), 'from', languages)
```

```
-------- output ---------
<class 'set'> {'SPSS', 'Ruby', 'SAS', 'Java', 'Julia', 'C++', 'C', 'AI', 'Data Science', 'R', 'Python'}
{'SPSS', 'Ruby', 'SAS', 'Java', 'Julia', 'C++', 'C', 'Data Science', 'R', 'Python'}
{'SPSS', 'Ruby', 'SAS', 'Java', 'Julia', 'C++', 'C', 'Data Science', 'R', 'Python'}
Removed:  SPSS from {'Ruby', 'SAS', 'Java', 'Julia', 'C++', 'C', 'AI', 'Data Science', 'R', 'Python'}
```

Operazioni sui set

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

# Set Union
print('Union of A | B', A|B) # Use | operator
print('Union of A and B', A.union(B)) # Alternative
```

```
-------- output ---------
Union of A | B {1, 2, 3, 4, 5, 6, 7, 8}
Union of A and B {1, 2, 3, 4, 5, 6, 7, 8}
```

```
# Set Intersection
print('Intersection of A & B', A&B) # Use & operator
print('Intersection of A and B', A.intersection(B))
```

```
-------- output ---------
Intersection of A & B {4, 5}
Intersection of A and B {4, 5}
```

```
# Set Difference
print('Difference of A - B', A-B) # Use - operator
print('Sym Difference of A and B', A.difference(B))
```

```
-------- output ---------
Difference of A - B {1, 2, 3}
Sym Difference of A and B {1, 2, 3}
```

Operazioni di base sui set (1)

```
languages ={'Python','R','SAS','Julia'}

# Return a shallow copy of a set
lang = languages.copy()
print("Languages : ", languages)
print("Lang : ", lang)

# Add an element in languages
languages.add('Java')
print("Languages : ", languages)
print("Lang : ", lang)
l=languages

# Add an element in l
l.add('C')
print("Languages : ", languages)
print("L : ", l)
```

```
-------- output ---------
Languages :  {'R', 'Python', 'SAS', 'Julia'}
Lang :  {'R', 'Python', 'SAS', 'Julia'}
```

```
-------- output ---------
Languages :  {'Julia', 'Java', 'SAS', 'R', 'Python'}
Lang :  {'R', 'Python', 'SAS', 'Julia'}
```

```
-------- output ---------
Languages :  {'C', 'Julia', 'Java', 'SAS', 'R', 'Python'}
L :  {'C', 'Julia', 'Java', 'SAS', 'R', 'Python'}
```

Operazioni di base sui set (2)

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

#Binary operations
print(A.isdisjoint(B)) #True for non intersecting sets
print(A.issubset(B))
print(A.issuperset(B))
```

```
------- output --------
False
False
False
```

```
#Unary operations
print("Sorting: ", sorted(A)) #Return a new sorted list
print("Sum: ", sum(A)) #Return the sum of all items
```

```
------- output --------
Sorting:  [1, 2, 3, 4, 5]
Sum:  15
```

```
print("length: ", len(A))
print("Min: ", min(A))
print("Max: ", max(A))
```

```
------- output --------
length:  5
Min:  1
Max:  5
```

Dizionari (dict)

3

Dizionari

- Il **dizionario** in Python avrà una **coppia chiave-valore** per ogni **elemento** che ne fa parte
- Le caratteristiche chiave del dizionario sono le seguenti
  - *La **chiave** e il **valore** devono essere racchiusi tra parentesi **graffe***
  - *Ogni **chiave** e **valore** sono separati utilizzando i due punti **[:]***
  - *Ogni **elemento** è separato da virgole **[,]***
  - *Le **chiavi** sono **univoche** all'interno di un dizionario specifico e devono essere tipi di dati **immutabili**: stringhe, numeri, tuple*
  - *I valori possono accettare **dati duplicati** di qualsiasi tipo*

Dizionari: accesso ed eliminazione

```
# Creating dictionary
dict={'Name':'Jivin','Age':6,'Class':'First'}
print('Sample dictionary: ', dict)

# Accessing items in dictionary
print('Value of key Name: ', dict['Name'])

# Example for deleting dictionary
del dict['Name'] # Delete specific item
print('Sample dictionary after deletion: ', dict)
dict.clear() # Delete all contents
print('Sample dictionary after clear: ', dict)
del dict # Delete the dictionary
```

```
-------- output ---------
Sample dictionary:  {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
Value of key Name:  Jivin
Sample dictionary after deletion:  {'Age': 6, 'Class': 'First'}
Sample dictionary after clear:  {}
```

Aggiornamento di un dizionario

```
dict={'Name':'Jivin','Age':6,'Class':'First'}
print('Sample dictionary: ', dict)

# Updating dictionary
dict['Age']= 6.5
print('Sample dictionary after updating: ', dict)
```

```
-------- output ---------
Sample dictionary:  {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
Sample dictionary after updating:  {'Name': 'Jivin', 'Age': 6.5, 'Class': 'First'}
```

Operazioni di base sui dizionari (1)

```
dict={'Name':'Jivin','Age':6,'Class':'First'}
print("length of dict: ", len(dict))
print("Equivalent string: ", str(dict))
```

```
-------- output ---------
length of dict:  3
Equivalent string:  {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
```

```
#Create a new dictionary with keys from tuple
tuple=('name','age','sex')
dict= dict.fromkeys(tuple)
print("New Dictionary: ", str(dict))
```

```
-------- output ---------
New Dictionary:  {'name': None, 'age': None, 'sex': None}
```

```
dict['name']='Jivin'
dict['age']= 7
dict['sex']='M'
print("New Dictionary: ", str(dict))
```

```
-------- output ---------
New Dictionary:  {'name': 'Jivin', 'age': 7, 'sex': 'M'}
```

```
dict= dict.fromkeys(tuple,10)
print("New Dictionary: ", str(dict))
```

```
-------- output ---------
New Dictionary:  {'name': 10, 'age': 10, 'sex': 10}
```

Operazioni di base sui dizionari (2)

```
dict={'Name':'Jivin','Age':6,'Class':'First'}
#Retrieve a value for a given key
print("Value for Age: ", dict.get('Age'))
print("Value for Sex: ", dict.get('Sex'))

""" Since the key Sex does not exist, the second
argument will be returned """
print("Value for Sex: ", dict.get('Sex', 'M'))
```

```
------- output --------
Value for Age:  6
Value for Sex:  None
Value for Sex:  M
```

Operazioni di base sui dizionari (3)

```
dict={'Name':'Jivin','Age':6,'Class':'First'}
# Check if key exists in dictionary
print("Age exists? ", 'Age' in dict)
print("Sex exists? ", 'Sex' in dict)
# Return items of dictionary
print("Dict items: ", dict.items())
# Return dictionary keys
print("Dict keys: ", dict.keys())
# Return values of dict
print("Dict values: ", dict.values())
```

```
-------- output ---------
Age exists?  True
Sex exists?  False
Dict items:  dict_items([('Name', 'Jivin'), ('Age', 6), ('Class', 'First')])
Dict keys:  dict_keys(['Name', 'Age', 'Class'])
Dict values:  dict_values(['Jivin', 6, 'First'])
```

Operazioni di base sui dizionari (4)

```
dict={'Name':'Jivin','Age':6,'Class':'First'}

""" if key does not exists, then the arguments will be added to
dict and returned"""
print("Value for Age: ", dict.setdefault('Age', None))
print("Value for Sex: ", dict.setdefault('Sex', 'M'))

# Concatenate dictionaries
dict={'Name':'Jivin','Age':6}
dict2={'Sex':'M'}
dict.update(dict2)
print("Concatenated dicts: ", dict)
```

```
-------- output ---------
Value for Age:  6
Value for Sex:  M
Concatenated dicts:  {'Name': 'Jivin', 'Age': 6, 'Sex': 'M'}
```

Funzioni e Generatori

4

Funzioni (1)

- Una **funzione** definita dall'utente è un **blocco** di **istruzioni** di codice correlate organizzate per ottenere una singola **azione correlata**
- L'obiettivo principale del concetto di funzioni definite dall'utente è incoraggiare la **modularità** e consentire la **riusabilità** del codice
- L'insieme di regole da seguire per definire una funzione in Python
  - La parola chiave **def** denota l'inizio di un **blocco funzione**, che sarà seguito dal **nome** della **funzione** e dalle **parentesi** di apertura/chiusura
  - I due punti **[:]** devono essere inseriti per indicare la **fine dell'intestazione** della funzione

Funzioni (2)

- **Sintassi**

```
def function_name():
    1st block line
    2nd block line
    ...
```

- L'insieme di regole da seguire per definire una funzione in Python
  - Le funzioni possono accettare argomenti o parametri
    - *Qualsiasi input di questo tipo deve essere inserito tra parentesi nell'intestazione del parametro*
  - Le istruzioni principali del codice devono essere inserite sotto l'intestazione della funzione e dovrebbero essere rientrate
    - *Per indicare che il codice fa parte della stessa funzione*

Funzioni (3)

- L'insieme di regole da seguire per definire una funzione in Python
- Le funzioni possono restituire un'espressione al chiamante
  - *Se il metodo **return** non viene utilizzato alla fine della funzione, agirà come una sottoprocedura*

```
# Simple function
def some_function():
    print("Hello World!")

# Call the function
some_function()
```

Funzioni con argomenti

- **Sintassi**

```
def function_name(parameters):
    1st block line
    2nd block line
    ...
    return [expression]
```

```
# Simple function to add two numbers
def sum_two_numbers(x,y):
    return x+y

# Call the function
print(sum_two_numbers(1,2))
```

Scope delle variabili

- La **disponibilità** di una **variabile** o **identificatore** all'interno del programma durante e dopo l'esecuzione è determinata dallo **scope** di una **variabile**
- Ci sono due ambiti variabili fondamentali in Python
  - ***Variabili globali***
  - ***Variabili locali***

```
x=10 # Global Variable

# Simple function to add two numbers
def sum_two_numbers(y):
    return x+y

# Call the function
print(sum_two_numbers(10))
```

Argomenti di default

- È possibile definire un **valore predefinito** per un argomento di funzione
  - la funzione **assumerà** o **utilizzerà** il valore **predefinito** nel caso in cui **non** venga **fornito** alcun valore nella chiamata alla funzione

```
# Simple function to add two numbers
def sum_two_numbers(x,y=10):
    return x+y

# Call the function
print(sum_two_numbers(10))
print(sum_two_numbers(10,5))
```

Elaborazione di più argomenti (1)

- Python ci consente di elaborare più argomenti di quelli specificati durante la definizione della funzione
  - *** args** e **** kwargs** sono un idioma comune per consentire un numero dinamico di argomenti
- ***args**

```
""" The *args will provide all function parameters in the
form of a tuple"""
# Simple function to loop through arguments
def sample_function(*args):
    for a in args:
        print(a)

# Call the function
sample_function(1,2,3)
```

Elaborazione di più argomenti (2)

- Python ci consente di elaborare più argomenti di quelli specificati durante la definizione della funzione
  - *** args** e **** kwargs** sono un idioma comune per consentire un numero dinamico di argomenti
- ***kwargs**

```
""" The **kwargs will give you the ability to handle named or
keyword arguments keyword that you have not defined in advance"""
# Simple function to loop through arguments
def sample_function(**kwargs):
    for a in kwargs:
        print(a, kwargs[a])

# Call the function
sample_function(name="John",age=27)
```

Funzioni annidate

- Python supporta il concetto di “**funzione annidata**”
  - È semplicemente una funzione definita all'interno di un'altra funzione

```
#main()
calculator(6, 5)
```

```
# Simple function to make calculus
def calculator(x,y):
    def sum(x,y):
        return x+y
    def sub(x,y):
        return x-y
    def mul(x,y):
        return x*y

    print(sum(x,y))
    print(sub(x,y))
    print(mul(x,y))
```

Generatori

- I **generatori** Python sono un modo semplice per creare iteratori
- Un generatore è una funzione che **restituisce** un **oggetto** (iteratore) su cui possiamo **iterare** (un valore alla volta)
  - È facile come definire una normale funzione con l'istruzione **yield** invece di un'istruzione
  - Se una funzione contiene almeno un'istruzione yield diventa una funzione generatrice
- **Sintassi**

```
def generator_function():
    yield [expression1]
    yield [expression2]
    …
```

Generatori vs Funzioni

- La differenza è
  - *un'istruzione **return** termina completamente una funzione*
  - ***yield** mette in pausa la funzione salvando tutti i suoi stati e poi continua da lì nelle chiamate successive*

- Una funzione di generatore differisce da una funzione normale
  - *La funzione **Generator** contiene una o più istruzioni di **yield***
  - *Le variabili locali e i loro stati vengono ricordati tra le chiamate successive*

Generatore di Fibonacci

```
x=10 # Global Variable

# Simple Fibonacci generator
def fib(n):
    a, b = 0, 1
    for i in range(n):
      yield a
      a, b = b, a+b

# main
print(list(fib(x)))
```

Fibonacci	Arithmetic
0	
1	
1	0 + 1 = 1
2	1 + 1 = 2
3	1 + 2 = 3
5	2 + 3 = 5
8	3 + 5 = 8
13	5 + 8 = 13
21	8 + 13 = 21
34	13 + 21 = 34

Perchè usiamo i Generatori?

- Esistono diversi motivi che rendono i generatori un'implementazione interessante da adottare
  - **Facile** da implementare
  - Memoria **efficiente**
  - Rappresenta il flusso **infinito**
  - **Pipeline** di Generatori
    - *I generatori possono essere utilizzati per convogliare una serie di operazioni*

Classi
Attributi e Metodi

5

Classi

- Una **classe** è un tipo di dati speciale che definisce come costruire un certo tipo di **oggetto**

- Le **istanze** sono **oggetti** creati che seguono la definizione data all'interno della **classe**

- Python non utilizza definizioni di interfaccia di classe separate come in alcuni linguaggi
  - *Devi solo definire la **classe** e poi usarla*

Metodi nelle classi

- Definire un metodo in una classe includendo le definizioni di funzione nell'ambito del **blok** di classe
- Deve esserci un primo argomento speciale **self** in tutte le definizioni di metodo che viene associato all'istanza chiamante
- Di solito c'è un metodo speciale chiamato **__init__** nella maggior parte delle classi

```
# A class representing a student
class student:

    def __init__(self,n,a):
        self.full_name=n
        self.age=a

    def get_age(self):
        return self.age
```

Istanziare Oggetti (1)

- **Non esiste una parola chiave "new" come in Java.**
- Basta usare il nome della classe con la notazione ( ) e assegnare il risultato a una variabile

```
# A class representing a student
class student:

    def __init__(self,n,a):
        self.full_name=n
        self.age=a

    def get_age(self):
        return self.age
```

```
b = studente( "Bob" , 21 )
```

Istanziare Oggetti (2)

- **__init__** funge da costruttore per la classe
- Gli argomenti passati al nome della classe vengono passati al suo metodo **__init__()**
- Un metodo **__init__** può accettare un numero qualsiasi di argomenti
- Come altre funzioni, gli argomenti possono essere definiti con valori predefiniti, rendendoli facoltativi per il chiamante

self

- Il **primo argomento** di ogni metodo è un riferimento all'istanza corrente della classe
  - *Per convenzione chiamiamo questo argomento **self***
- Sebbene sia necessario specificare **self** esplicitamente quando si definisce il metodo, non lo si include quando si chiama il metodo.
  - *Python te lo passa automaticamente*

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
  self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Come accedere ad attribute e metodi?

```
>>> f = student("Bob Smith", 23)

>>> f.full_name # Access attribute
"Bob Smith"

>>> f.get_age() # Access a method
23
```

Attributi (1)

- Esistono due tipi di attributi
  - **Attributi dei dati**
    - Variabile posseduta da una particolare istanza di una classe
    - Ogni istanza ha il suo valore
    - Questi sono il tipo più comune di attributo
    - Gli attributi dei dati vengono creati e inizializzati da un metodo __init__()
  - **Attributi di classe**
    - Di proprietà della classe nel suo complesso
    - Tutte le istanze della classe condividono lo stesso valore
    - Chiamate variabili “static” in alcune lingue
    - Accedi agli attributi della classe utilizzando **self.__class__.name** notazione

Attributi (2)

```
class counter:
  overall_total = 0
        # class attribute
  def __init__(self):
      self.my_total = 0
        # data attribute
  def increment(self):
     counter.overall_total = \
     counter.overall_total + 1
     self.my_total = \
     self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

Metodi Speciali

- Esistono metodi che esistono per tutte le classi
  - *Puoi sempre ridefinirli*

- Esempi di metodi speciali
  - **__init__** : il costruttore della classe
  - **__cmp__** : Definisce come == funziona per la classe
  - **__len__** : Definisce come funziona **len(obj)**
  - **__copy__** : Definisce come copiare una classe
  - **__repr__** : Definisce come trasformare un'istanza in una stringa

Elementi Speciali

- Esempi di dati speciali
  - **__doc__** : Variabile per le stringhe di documentazione per la classe
  - **__class__** : Variabile che ti dà un riferimento alla classe da qualsiasi sua istanza

```
>>> f = student("Bob Smith", 23)

>>> print f.__doc__
A class representing a student.

>>> f.__class__
< class studentClass at 010B4C6 >
```

Dati privati e metodi

- Qualsiasi attributo/metodo con due caratteri di sottolineatura iniziali nel nome (ma nessuno alla fine) è privato e non è possibile accedervi al di fuori della classe

- Nota: i nomi con due caratteri di sottolineatura all'inizio e alla fine si riferiscono a metodi o attributi incorporati per la classe Variabile posseduta da una particolare istanza di una classe

```
class MyClass:
    def myPublicMethod(self):
            print 'public method'
    def __myPrivateMethod(self):
            print 'this is private!!'
```

Tips

Suggerimenti (1)

- Importa un modulo

```
# Import all functions from a module
from module_name import *

# Import a specific function from a module
from module_name import function_name
```

Suggerimenti (2)

- La gestione delle eccezioni

```
# Below code will open a file
fName="vechicles.txt"
try:
    with open(fName, 'r') as f
    print(f.readline())
except IOError as e:
    print('IO Error')
finally:
    print('File has been closed')
```