

Tecnologie Cloud e Mobile

Lez. 11

Python

Giuseppe Psaila

Università di Bergamo

giuseppe.psaila@unibg.it

Introduzione

Python

- È stato progettato all'inizio degli anni '90 in Olanda
- È un linguaggio relativamente semplice, con tipizzazione dinamica, interpretato e a oggetti
- È diventato molto popolare perché polivalente
- È utilizzato moltissimo nell'ambito della Data Science, perché esistono moltissime librerie per gli usi più svariati

Sito Ufficiale

- `https://www.python.org/`
- Scegliendo «Downloads», si può scaricare la versione per il proprio sistema operativo
- Al momento dell'installazione, attuavate l'opzione «*Add Python 3.X to PATH*». Se l'installer non lo chiede, verificate che non venga segnalato qualche problema con il path.

[Donate](#)[GO](#)[Socialize](#)[About](#)[Downloads](#)[Documentation](#)[Community](#)[Success Stories](#)[News](#)[Events](#)

```
# Python 3: Simple arithmetic
>>> 1 / 2
0.5
>>> 2 ** 3
8
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>> 17 // 3 # floor division
5
```

>_

Intuitive Interpretation

Calculations are simple with Python, and expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work as expected; parentheses `()` can be used for grouping. [More about simple math functions in Python 3.](#)

[1](#)[2](#)[3](#)[4](#)[5](#)

Python is a programming language that lets you work quickly and integrate systems more effectively. [>>> Learn More](#)

Interprete

- L'interprete Python è a linea di comando, quindi si presenta così

```
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit  
(Intel)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> a=1  
>>> b=a  
>>> print(a)  
1  
>>>
```

Modalità Operativa

- Il prompt «>>» è in attesa di un'istruzione
- Inserita un'istruzione, si ripresenta il prompt
- Il programma eseguito è

```
a=1
b=a
print(a)
```
- Ovviamente, così è scomodo, meglio usare un IDE

Quale IDE?

- Ce ne sono tantissimi
- **IDLE** è installato insieme a Python, ma è troppo semplice e limitato
- **Spyder** è molto completo, useremo questo
- **Jupyter** crea un ambiente web locale, quindi si lavora in una pagina web

Spyder

- Pagina di download
`https://www.spyder-ide.org/`
- Spyder ha un buon debugger

```
1 prev = 1
2 print(prev)
3 current = 1
4 print(current)
5 while current < 20:
6     next_val = current + prev
7     prev = current
8     current = next_val
9     print(next_val)
10
```

Name	Type	Size	Value
------	------	------	-------

Variable explorer Help Plots Files

```
ipdb> !runfile('C:/Users/Utente/Documents/Lavoro/corsi/
Cloud_Mobile/Python/untitled5.py', wdir='C:/Users/Utente/
Documents/Lavoro/corsi/Cloud_Mobile/Python')
21
```

ipdb>

In [22]:

Removing all variables...

In [22]:

IPython console History

Spyder

- Il programma calcola un pezzo della serie di Fibonacci
- Premendo sul pulsante indicato, esegue il programma nella finestra corrente, mette il risultato in basso a destra e in alto a destra sono mostrate le variabili

```
1 prev = 1
2 print(prev)
3 current = 1
4 print(current)
5 while current < 20:
6     next_val = current + prev
7     prev = current
8     current = next_val
9     print(next_val)
10
```

Name	Type	Size	Value
current	int	1	21
next_val	int	1	21
prev	int	1	13

Variable explorer Help Plots Files

```
In [22]: runfile('C:/Users/Utente/Documents/Lavoro/corsi/
Cloud_Mobile/Python/fibonacci.py', wdir='C:/Users/Utente/
Documents/Lavoro/corsi/Cloud_Mobile/Python')
```

```
1
1
2
3
5
8
13
21
```

```
In [23]:
```

IPython console History

Spyder

- Spyder include già l'interprete Python, quindi non serve scaricarlo
- Ma se si vuole usare una versione diversa da quella di Spyder, si può configurare

Spezzare il Codice

- Spezziamo il codice su due schede
- Il ciclo lo mettiamo nella seconda scheda
- Poi eseguiamo il primo pezzo del programma

Seconda parte

- Andando nella seconda scheda, se premiamo il solito pulsante di esecuzione, viene dato errore, perché le variabili usate non sono definite
- Ma premendo il pulsante alla destra della freccia verde, il frammento viene eseguito correttamente: usa le variabili già esistenti



C:\Users\Utente\Documents\Lavoro\corsi\cloud_Mobile\Python\untitled6.py

untitled3.py × fibonacci.py × **untitled6.py*** ×

```
1 while current < 20:
2     next_val = current + prev
3     prev = current
4     current = next_val
5     print(next_val)
6
```

Variable explorer

Console 1/A

```
In [35]: runcell(0, 'C:/Users/Utente/Documents/Lavoro/corsi/Cloud_Mobile/Python/untitled6.py')
```

2
3
5
8
13
21

In [36]:

IPython console

Esecuzione a Pezzi

- L'esecuzione a pezzi si può ottenere anche creando le «cells», con i caratteri «#%%»
- Lo stesso pulsante usato prima esegue una cella
- Il pulsante alla sua destra esegue una cella e poi attiva la successiva
- Così facendo, si può osservare che cosa fa una cella, prima di procedere con i pezzi successivi

C:\Users\Utente\Documents\Lavoro\corsi\Cloud_Mobile\Python\fibonacci.py

untitled3.py × fibonacci.py ×

```
1 prev = 1
2 print(prev)
3 current = 1
4 print(current)
5
6 while current < 20:
7     next_val = current + prev
8     prev = current
9     current = next_val
10    print(next_val)
```

Name	Type	Size	Value
current	int	1	1
prev	int	1	1

Variable explorer Help Plots Files

Console 1/A ×

Cloud_Mobile/Python/fibonacci.py')

1
1

In [37]:

IPython console History

C:\Users\Utente\Documents\Lavoro\corsi\Cloud_Mobile\Python\fibonacci.py

untitled3.py x fibonacci.py

```
1 prev = 1
2 print(prev)
3 current = 1
4 print(current)
5 #%%
6 while current < 20:
7     next_val = current + prev
8     prev = current
9     current = next_val
10 print(next_val)
```

Name	Type	Size	Value
current	int	1	21
next_val	int	1	21
prev	int	1	13

Variable explorer Help Plots Files

Console 1/A x 

```
(Cloud_Mobile/Python/fibonacci.py)
1
1

In [37]: runcell(1, 'C:/Users/Utente/Documents/Lavoro/corsi/
Cloud_Mobile/Python/fibonacci.py')
2
3
5
8
13
21

In [38]:
```

Python console History

Debugging

- Cliccando vicino al numero di linea, un pallino rosso indica che un breakpoint è stato impostato
- Avviando l'esecuzione in modalità debugging, si può ispezionare l'esecuzione passo passo

C:\Users\Utente\Documents\Lavoro\corsi\Cloud_Mobile\Python\fibonacci.py

untitled3.py × fibonacci.py ×

```
1 prev = 1
2 print(prev)
3 current = 1
4 print(current)
5 #%%
6 while current < 20:
7     next_val = current + prev
8     prev = current
9     current = next_val
10    print(next_val)
```



Name	Type	Size	Value
current	int	1	1
prev	int	1	1

Variable explorer Help Plots Files

Console 1/A × 

```
3
5
8
13
21

In [38]: debugfile('C:/Users/Utente/Documents/Lavoro/corsi/
Cloud_Mobile/Python/fibonacci.py', wdir='C:/Users/Utente/
Documents/Lavoro/corsi/Cloud_Mobile/Python')

1
1

ipdb> |
```

IPython console History

Il Linguaggio

Caratteristiche

- Linguaggio interpretato
- Type checking dinamico
- Sintassi snella
- Le variabili non vanno dichiarate
- Le variabili sono puntatori/reference (come in Java)
- Programmazione a oggetti
- I tipi built-in sono classi, quindi i valori sono oggetti

Tipi delle variabili

- Numerici
 - int
 - float
 - complex
- Liste
 - string
 - list
 - tuple
- Dizionari: Mappe chiave-valore

Numeri

- Classi/Costruttori/Casting
 - `int(...)`
 - `float(...)`
- Operatori Aritmetici
 - `+`
 - `-`
 - `*`
 - `/` divisione con parte frazionaria
 - `//` quoziente
 - `%` resto
 - `**` potenza: `3**2` è 3^2

Numeri

- Operatori Incrementali

- +=

- -=

- *=

- /= divisione con parte frazionaria

- // = quoziente

- %= resto

- **= potenza (esponente a destra)

Numeri

- Esempio

```
a = int("12")  
b = float( a )
```

Nam	Type	Size	
a	int	1	12
b	float	1	12.0

- Il costruttore degli interi converte la stringa in intero
- Il costruttore dei float converte l'intero in float

Numeri complessi

- `a = complex(1, 2)`
`print(a)`
`print(a.real)`
`print(a.imag)`
- Stampa:
`(1+2j)`
`1.0`
`2.0`

Costruttore
Stampa sulla console

Oggetti Mutable/Immutable

- Immutable
 - Gli oggetti (perché le variabili puntano agli oggetti), una volta creati, non possono essere cambiati
 - Numeri, stringhe, tuple
- Mutable
 - Gli oggetti possono cambiare il loro stato: esistono metodi di modifica
 - Liste, dizionari

Stringhe

- Non hanno un costruttore
- Semplicemente si assegna una stringa
`a = "Ciao"`
- Come convertire un numero?
`b = "{}".format(12)`
- Il metodo **format** sostituisce le graffe con il valore dell'argomento
- Il metodo **format** accetta un numero variabile di parametri (tanti quante sono le parentesi graffe nella stringa)

Nam	Type	Size	
b	str	1	12

Stringhe

- Python fornisce molti metodi per manipolare le stringhe

- Non sto a riportarli, fate riferimento a

`https://www.w3schools.com/python/python_ref_string.asp`

Liste

- Non hanno un costruttore
- Sostituiscono i vettori/array

```
l = [ 1, "b"]  
l.append( 3 )  
print( l[2] )  
print( l )
```

Var	Type	Size	
l	list	3	[1, 'b', 3]

- La lista viene stampata con le quadre

```
3  
[1, 'b', 3]
```

Liste

- Python fornisce molti metodi per manipolare le liste

- Non sto a riportarli, fate riferimento a

`https://www.w3schools.com/python/python_ref_list.asp`

Tuple

- Ha il costruttore tuple(...)
- Sono dette liste immutabile (ma sono un po' rognose)

```
t = (1, "b")
t1 = tuple("a")
t2 = t + t1
print( t2[2] )
print( t2 )
```

Narr ▲	Type	Size	
t	tuple	2	(1, 'b')
t1	tuple	1	('a')
t2	tuple	3	(1, 'b', 'a')

```
a
(1, 'b', 'a')
```

- Se si assegna `t1 = ("a")`, `t1` è una stringa

Tuple

- Python fornisce molti metodi per manipolare le tuple
- Non sto a riportarli, fate riferimento a https://www.w3schools.com/python/python_tuples.asp

Stringhe, Liste, Tuple

- Hanno in comune alcuni costrutti e funzioni
- `len(o)` lunghezza
- `o[2]` terzo elemento
- `o[1:3]` elementi da 1 a 2 (3 escluso)
- `o[-1]` ultimo elemento

Set e Frozenset

- Esistono anche le collezioni di tipo **set**, cioè senza ripetizioni
 - **Set** insieme mutable
 - **Frozenset** Insieme non mutable
- Sono meno integrati nel linguaggio delle liste e meno usati (non vediamo esempi)

Dizionari

- Detti «mappe» in altri contesti
- Sono contenitori chiave-valore
- Hanno il costruttore `dict()`, ma si può usare anche `{}`
- Se vi ricorda JSON, è proprio così: le costanti dei dizionari sono proprio dei documenti JSON

Dizionari

- ```
d1 = {}
d1["name"] = "Pippo"
d1["age"] = 25
d1["cars"] = [{"Model": "A"}, {"Model": "C"}]
d2 = {"name": "Pluto", "age": 30}
```

| Narr ▲ | Type | Size | Value                                                             |
|--------|------|------|-------------------------------------------------------------------|
| d1     | dict | 3    | <code>{'name': 'Pippo', 'age': 25, 'cars': [{...}, {...}]}</code> |
| d2     | dict | 2    | <code>{'name': 'Pluto', 'age': 30}</code>                         |



# Variabili come Puntatori/Reference

- Usando gli oggetti mutable, si capisce come le variabili siano puntatori/reference agli oggetti

- Esempio


```
l1 = [1, 2]
```

```
l2 = l1
```

```
l1.append(3)
```

- Se fossero diverse, l2 non conterrebbe 3

| Narr | Type | Size | Value     |
|------|------|------|-----------|
| l1   | list | 3    | [1, 2, 3] |
| l2   | list | 3    | [1, 2, 3] |



# Variabili come Puntatori/Reference

- L'operatore «+» concatena le liste, quindi ne crea una nuova

- Esempio

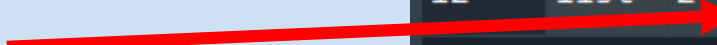
```
l1 = [1, 2]
```

```
l2 = l1
```

```
l1 = l1 + [3]
```

- La lista l1 è cambiata, ma non l2

| Narr ▲ | Type | Size | Value     |
|--------|------|------|-----------|
| l1     | list | 3    | [1, 2, 3] |
| l2     | list | 2    | [1, 2]    |



# Istruzioni di Controllo

- Sono le solite
  - if ... elif .... else (elif sta per elseif)
  - while
  - for
- Ma non si usano parentesi: l'annidamento si fa con l'indentazione, usando il «tab»
- La condizione è seguita da «:»

# Istruzione Condizionale

- La presentiamo con un esempio

- `a=2`

```
if a==1:
 print("A")
elif a==2:
 print("Vale")
 print(a)
else:
 print("Nessuno")
```

L'azione  
di elif

# while

- Sintassi:
- *while condizione:*  
    azione (righe indentate allo stesso modo)
- È un while classico, quindi la condizione viene valutata prima di entrare nel ciclo

# while

- Esempio: somma primi 10 numeri

```
somma = 0
```

```
n=1
```

```
while n <= 10:
```

```
 somma += n
```

```
 n +=1
```

```
print("Somma: {}".format(somma))
```

Somma: 55

# for

- Sintassi  
for var in lista:  
    azione
- Quindi, non è un ciclo a contatore, ma **a iteratore**
- La variabile specificata itera sulla lista

# for

- Esempio: lunghezza media di una lista di stringhe

```
lista = ["a", "aa", "aaa"]
```

```
somma=0
```

```
for s in lista:
```

```
 somma += len(s)
```

```
print("Media: {}".format(somma /
len(lista)))
```

```
Media: 2.0
```



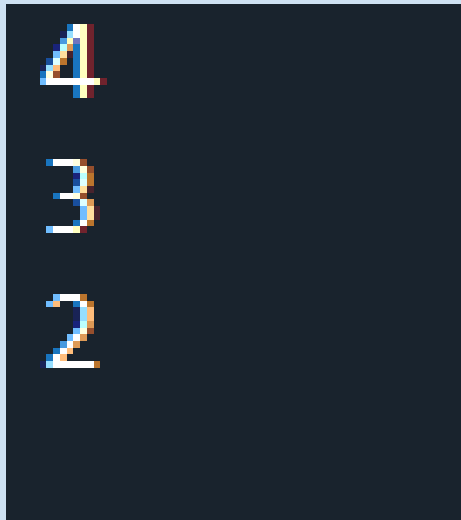
# for e cicli a contatore

- Quindi i cicli a contatore si possono fare solo con while?
- No, esiste una funzione **range** che ritorna una lista con tutti i valori in un intervallo
- **range(from, to, step)**
  - from:     valore iniziale (incluso)
  - to:        valore finale (escluso)
  - step:     incremento (opzionale, anche negativo)

# for e cicli a contatore

- Esempio

```
for v in range(4, 1, -1):
 print(v)
```



4  
3  
2

# Uscire/Continuare il Ciclo

Come in C (e nei linguaggi derivati)

- break                      esce dal ciclo
- continue                  passa all'iterazione successiva

# **else dopo i cicli**

- Si può specificare un ramo else dopo i cicli
- Viene eseguito se il ciclo termina perché la condizione è falsa
- L'uscita con break salta anche il ramo else

# else dopo i cicli

- Esempio:
- ```
lista = ["a", "aa", "aaa"]
somma=0
for s in lista:
    somma += len(s)
else:
    print("Media: {}".format(somma /
len(lista)))
```

Operatori di Confronto

- == uguaglianza (dei valori, non dei reference)
- != diversità (dei valori)
- >
- >=
- <
- <=

Operatori di Confronto

- Esempio:

```
a = [1]
```

```
b = [1, 2]
```

```
a.append(2)
```

```
if a==b:
```

```
    print("uguali")
```



uguali

- Sono chiaramente due oggetti diversi, ma il confronto per uguaglianza è vero

Operatori Logici

- and
- or
- not

Input

- Esiste una funzione **input** che legge da tastiera e restituisce la stringa letta.
- Il parametro è il testo da mostrare per richiedere l'input
- Esempio:

```
x = input('Enter your name: ')\nprint('Hello, ' + x)
```

Input

- Il parametro è opzionale
- Esempio:

```
print('Enter your name: ')\n $x$  = input()\nprint('Hello, ' +  $x$ )
```

Definizione di Funzioni

- Sintassi
- `def nomefunzione(parametri):`
 codice (con tab)

- Esempio

```
def f(x, y):  
    m = (x + y) / 2  
    return m
```

```
print( f(3, 6) )
```

Funzioni Annidate

- Le funzioni possono essere definite dentro altre funzioni

- Esempio

```
def f1(a):  
    def f2(b):  
        return b+1  
    return f2(a)
```

```
print( f1(3) )
```

- f2 è visibile solo in f1

Visibilità delle variabili

- La funzione vede tutte le variabili definite all'esterno
- Ma una variabile nella funzione può coprire una variabile esterna
- Esempio:

```
pi = 'outer pi variable'
def print_pi():
    pi = 'inner pi variable'
    print(pi)
print_pi()
print(pi)
```

```
inner pi variable
outer pi variable
```

Visibilità delle Variabili

- L'accesso alla variabile esterna è in sola lettura
- Se prima si usa la variabile esterna e poi si cerca di cambiarla, viene segnalato un errore

```
10 pi = 'outer pi variable'
11 def print_pi():
12     print(pi)
13     pi = 'inner pi variable'
14 print_pi()
15 print(pi)
16
```

Variabili Esterne Mutable

- Le variabili esterne non sono modificabili da una funzione
- Ma se il valore è un oggetto mutable, lo stato dell'oggetto può essere cambiato (senza cambiare il valore della variabile, cioè il reference all'oggetto)

```
• elenco = []  
def f( valore ) :  
    elenco.append(valore)
```

```
f(1)
```

Nome	Type	Size	
elenco	list	1	[1]

Visibilità delle Variabili

- Una variabile locale può essere dichiarata visibile a livello globale
- Si dichiara la variabile «global»

```
• pi = 'outer pi variable'
  def f():
      global internal_pi
      internal_pi = 'inner pi variable'
  f()
  print(pi)
  print(internal_pi)
```

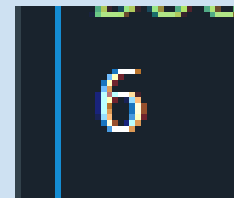
```
outer pi variable
inner pi variable
```


Parametri Formali

- La lista dei parametri formali può essere vuota
- I parametri sono separati da virgola
- I parametri in coda possono avere un valore di default, quindi si possono non specificare
- Esempio:

```
def f(a, b, c=3):  
    return a + b + c
```

```
print(f(1, 2))
```



Parametri Formali

- Se i parametri con valore di default sono multipli, si può scegliere a quali passare il valore

- Esempio:

```
def f(a, b, c=3, d=0, e=0):  
    return a + b + c + d + e
```

```
print(f(1, 2, d=1, e=1))
```



8

Parametri Formali Illimitati

- Il parametro ***varargs** indica una lista anche vuota di parametri attuali. `varargs` è una tupla

```
def f(a, b, *varargs):  
    s = 0  
    for v in varargs:  
        s += v  
    return a + b + s
```

```
print(f(1, 2, 1, 1))  
print(f(1, 2, 1, 1, 3))
```



5
8

Classi

- Introduciamo la programmazione a Oggetti con un esempio
- Una classe «Archivio» serve per gestire insiemi di nominativi
- Fornisce tre metodi:
 - **add_nome** per inserire un nuovo nome
 - **show** per mostrare i nomi
 - **clean** svuota l'elenco

Classi

```
class Archivio:
    def __init__(self):
        self.elenco = []
        return
    def add_nome(self, nome, eta):
        self.elenco.append({"nome": nome, "eta": eta})
        return
    def show(self):
        for e in self.elenco:
            print(e)
        return
    def clean(self):
        self.elenco = []
        return
```

Classi

- La parola chiave «class» introduce la definizione della classe
- Il contenuto della classe è indentato dopo il «:»
- I metodi sono definiti quasi come normali funzioni: obbligatorio il parametro «self», che fa riferimento all'oggetto
- Al momento della chiamata è implicitamente inizializzato
- Serve per accedere ai campi dell'oggetto

Utilizzo della classe «Archivio»

```
nomi= Archivio()  
nomi.add_nome("Pippo", 25)  
nomi.add_nome("Pluto", 30)  
nomi.show()
```

- Per creare l'oggetto, basta chiamare il costruttore

```
{ 'nome': 'Pippo', 'eta': 25 }  
{ 'nome': 'Pluto', 'eta': 30 }
```

Costruttori

- Per creare uncostruttore
`def __init__(self e eventuali parametri) :`
- Che può aggiungere i campi all'oggetto, con `self.campo`
- Ogni metodo può aggiungere campi all'oggetto
- **È ammesso un solo costruttore**

Ereditarietà

- È possibile definire delle sotto-classi
- Sintassi:
`class sottoclasse(superclasse) :`
- La sotto-classe eredita automaticamente campi e metodi
- La sotto-classe può fare l'overriding dei metodi ereditati

Chiamare il Costruttore della Super-classe

- `super()` fa riferimento alla super-classe
- Consente di chiamare il costruttore della super-classe:
`super().__init__(parametri)`

Esempio: Super-classe Figura

```
class Figura:
    def __init__(self, id, tipo):
        self.tipo = tipo
        self.id = id
        return
    def che_tipo(self):
        return self.tipo
    def get_area(self):
        return 0
```

Esempio

- Nell'esempio, viene definita la super-classe Figura
- Il costruttore riceve due parametri:
 - `id` un identificatore
 - `tipo` il tipo di figura
- Fornisce due metodi
 - `che_tipo` restituisce il tipo di figura
 - `get_area` restituisce l'area (sempre 0)

Esempio: Sotto-classe Rettangolo

```
class Rettangolo(Figura):  
    def __init__(self, id, base, altezza):  
        super().__init__(id, "Rettangolo")  
        self.area = base * altezza  
        return  
  
    def get_area(self):  
        return self.area
```

Esempio: Sotto-classe Rettangolo

- Il costruttore chiama il costruttore della super-classe e inizializza il cmpo «area»
`super().__init__(id, "Rettangolo")`
`self.area = base * altezza`
- Ridefinisce il metodo «get_area»

Esempio: Sotto-classe Triangolo

```
class Triangolo(Figura):  
    def __init__(self, id, base, altezza):  
        super().__init__(id, "Triangolo")  
        self.area = base * altezza / 2  
        return  
    def get_area(self):  
        return self.area
```

Esempio: Sotto-classe Triangolo

- Il costruttore chiama il costruttore della super-classe e inizializza il cmpo «area»
`super().__init__(id, "Triangolo")`
`self.area = base * altezza / 2`
- Ridefinisce il metodo «get_area»

Esempio: Uso

```
elenco = []  
elenco.append( Rettangolo(1, 2, 1))  
elenco.append( Triangolo(2, 2, 1))  
  
for f in elenco:  
    print(f.che_tipo())  
    print( f.get_area())
```

Esempio: Uso

- Si crea una lista vuota
- Alla lista vengono aggiunti due oggetti, uno di tipo «Rettangolo» e l'altro di tipo «Triangolo»
- Il ciclo for scandisce l'elenco e, per ogni elemento
 - Stampa il valore restituito dal metodo «che_tipo» (della super-classe)
 - Stampa il valore restituito dal metodo «get_area» (ridefinito dalla sotto-classe)

```
Rettangolo  
2  
Triangolo  
1.0
```

Eccezioni

- Sintassi
- try:
 - codice monitorato
 - except eccezione:
 - reazione
 - except ...
 - else: # opzionale
 - codice da eseguire se non ci sono eccezioni

Eccezioni: Esempio

```
op1 = int(input("Dividendo:"))
op2 = int(input("Divisore"))
try:
    n = op1 / op2
except ZeroDivisionError as err:
    print('Invalid operation ({})!'.format(err))
except ArithmeticError:
    print('Invalid operation!')
else:
    print("Risultato: {}".format(n))
```

Eccezioni: Esempio

- In caso di divisione per zero
- Entrambe le eccezioni sono valide: il primo ramo except viene eseguito
- L'alias «as err» consente di ottenere il messaggio di errore dell'eccezione
- Se «op2» non vale zero, viene eseguito il ramo else

Generare Eccezioni

- L'istruzione «raise» consente di generare le eccezioni
- `raise ZeroDivisionError('Impossibile dividere per 0')`
- Si può anche propagare l'eccezione catturata
`except ZeroDivisionError as err:`
`raise err`
- Le eccezioni sono classi
- Sono 30 eccezioni built-in, le trovate qui:
`https://www.programiz.com/python-programming/exceptions`

Definire Eccezioni

- Si possono definire eccezioni, come sotto-classi della classe «Exception»

- Esempio

```
class MyException(Exception):  
    pass
```

```
raise MyException("CIAO")
```

- L'istruzione «pass» consente di lasciare vuoti i blocchi annidati

```
MyException: CIAO
```

Garbage Collector?

- Il comportamento di Python nella gestione degli oggetti ricorda quello di Java
- Non esiste una de-allocazione esplicita
- Quindi, anche Python ha il «garbage collector»
- Nella prossima lezione, parleremo di moduli: il modulo «gc» consente di gestire la memoria e il garbage collector

Conclusioni

- Ci manca un concetto importante: i moduli
- Lo vedremo nella prima parte della prossima lezione, con un esempio concreto che discuteremo
- Nel quale i moduli saranno fondamentali
- Ma conoscendo la programmazione a oggetti (Java) e l'approccio dinamico al type checking (JavaScript), Python è veramente veloce da imparare