Corso di **Python**

Lezione 2

I tipi di dati

Editor: Davide Brunato
Scuola Internazionale Superiore di Studi Avanzati di Trieste

Variabili

- Una **variabile** inizia ad esistere quando si assegna un valore ad un **nome**:

```
a = 10
```

- Dinamicamente si può cambiare valore e tipo:

```
>>> a = 10.1
>>> a = "Stringa"
```

- Per convenzione gli identificatori delle variabili sono scritti in minuscolo, usando l'underscore per separare le parole:

```
contatore_linee += 1
```

Costanti

- In Python sono definite solo un numero limitato di costanti built-in:
  - **False**: valore booleano *falso*
  - **True**: valore booleano *vero*
  - **None**: valore NoneType, che indica nessun oggetto/valore
  - **NotImplemented**: ritornato quando l'operatore binario non è definito
- In Python3 i nomi delle costanti built-in sono anche keyword non riassegnabili
- In Python, a parte quelle predefinite dal linguaggio e i valori espressi direttamente come letterali, non esistono costanti nel senso classico
- In generale le **costanti** sono oggetti identici alle variabili, solo che per convenzione vengono assegnate una volta sola e il nome contiene solo caratteri maiuscoli ed underscore:

```
LANGUAGE_CODE = 'it-IT'
```

Mutabili e immutabili

- Si parla di oggetto **mutabile** quando il suo valore può essere alterato
- Un oggetto che non può essere alterato dopo la creazione è detto invece **immutabile**
- Mutabilità dei tipi di dati in Python:
  - I tipi semplici (numerici, stringhe e bytes) sono tutti immutabili
  - I tipi strutturati sono mutabili (liste, insiemi, dizionari), ad esclusione di un paio (*tuple* e *frozenset*)

Oggetti hashable

- In Python si parla di oggetto **hashable** quando è mappabile su un valore numerico che non cambia per tutta la sua esistenza
- In generale tutti gli oggetti immutabili sono hashable, mentre i tipi di dato strutturati mutabili non lo sono
- Per ottenere il valore di hash di un oggetto esiste apposita funzione:

```
>>> hash("Stringa di test")
-3932773316872885800
>>> s = "Stringa di test"
>>> hash(s)
-3932773316872885800
>>> hash(1999)
1999
```

- Il valore di hash è importante per la verifica di uguaglianza tra oggetti diversi
- Gli oggetti hashable sono importanti per la gestione di insiemi e dizionari

La funzione **print**

- La funzione **print** stampa su output valori in forma di testo
  - In Python 2 era un'istruzione che prevedeva parentesi, ma la sua forma ed il suo uso erano assimilabili ad una funzione
  - In Python 3 è stata spostata tra le funzioni built-in e pertanto prevede le parentesi tonde
  - Python 2.6 e 2.7 accettano entrambe le forme, mentre Python 3 accetta solo quella nuova
- Quando si esegue una *print* gli argomenti sono implicitamente o esplicitamente convertiti in formato testuale
- Può servire negli script CLI o come debug
- Esempi:

```
>>> i = 10
>>> print(i)
10
>>> print("Numero di iterazioni: %d" % i)
Numero di iterazioni: 10

>>> a, b, c, d = "Bene", 1000, False, None
>>> print(a, b, c, d)
('Bene', 1000, False, None)
>>> print("%s %s %s %s" % (a, b, c, d))
Bene 1000 False None
```

La funzione **dir**

- Visualizza gli attributi di un oggetto (funzione, classe, variabile, ...)
- Tutti gli oggetti di Python possiedono attributi:

```
>>> a = 1
>>> dir(a)
['__abs__', '__add__', '__and__', '__class__', '__cmp__',
'__coerce__', ..... 'bit_length', 'conjugate',
'denominator', 'imag', 'numerator', 'real']
```

anche la costante predefinita *None* ha parecchi attributi:

```
>>> dir(None)
['__class__', '__delattr__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
```

La funzione **type**

- Visualizza il tipo dell'oggetto passato come parametro:

```
>>> type(None)
<type 'NoneType'>
>>> type(False)
<type 'bool'>
>>> type(dir)
<type 'builtin_function_or_method'>
>>> type(10)
<type 'int'>
```

- La funzione type è usata da sola nelle sessioni interattive o in fase di debug mentre nel codice generalmente viene usata insieme alla funzione *isinstance*
- *La funzione type, quando chiamata con più parametri, permette di costruire nuovi tipi di oggetti ...*

Tipi di dati semplici

- Interi
- Floating-point
- Complessi
- Stringhe
- None
- Booleani

Tipi di dati numerici interi

- Numeri Interi (**int**, **long**)
  - Decimali: `1, 30, 1999`
  - Ottali: `01, 036, 03717` (da Python 2.6+: `0o1, 0o36, 0o3717`)
  - Esadecimali: `0x1, 0x1E, 0x7CF`
  - Binari: `0b11, 0b1010` (da Python 2.6+)
- In Python 2:
  - Il tipo **int** ha valore limite pari a *sys.maxint*, ma l'interprete trasforma automaticamente un valore in *long* quando il valore assoluto del numero è troppo elevato per un *int*
  - Il tipo **long** può essere esplicitato con suffisso '`L`' o '`l`' sul letterale
- In Python 3 il tipo *long* è stato eliminato, si usa solo *int*
- Dalla versione 2.2 i numeri interi di Python non hanno limiti:
  - Evita gli errori di overflow degli interi
  - Per numeri interi con overflow usare librerie dedicate (numpy)

Altri tipi di dati numerici

- Numeri Floating-point (**float**)
  - `0., 0.0, .0, 1., 1.0, 1e0, 1.e0., 1.0e0, 1.0E0, 1.0E-1`
  - Limiti e caratteristiche: `import sys; sys.float_info`
- Numeri Complessi (**complex**)
  - Coppia formata da due numeri floating-point: **z.real** e **z.imag**
  - `0j, 0.j, 0.0j, .0j, 1j, 1.j, 1.0j, 1e0j, 1.e0j, 1.0e0j`
- Da Python 2.4 è stato introdotto il tipo **Decimal** per operazioni con numeri decimali a precisione finita:

```
>>> import decimal
>>> d1 = decimal.Decimal('3.14'); d1
Decimal('3.14')
>>> d2 = decimal.Decimal(3.14); d2
Decimal('3.140000000000001243449787580175325274467468261718 75')
>>> print(d1 ** 2, d2 ** 2)
9.8596 9.859600000000000780886466600
```

Stringhe

- Sono sequenze immutabili di caratteri
- Per specificare un letterale stringa si possono usare gli apici o i doppi apici:

```
"Questa è una stringa"
'Questa è un\'altra stringa'  # con sequenza di escape
```

- Il tipo stringa è codificato come **str**:

```
>>> type("Questa è una stringa")
<type 'str'>
```

- Nei letterali stringa possono essere usate delle sequenze di escape:

\\	\'	\”	\a	\b	\f	\n
\r	\t	\v	\DDD	\xXX	\other	\<newline>

Bytes

- bytes
  - Sequenza *immutabile* di interi 0 <= x < 256
  - Introdotti perché da Python 3 tutte le stringhe sono per default Unicode
  - Bufferizzabili (accesso alla memoria senza copia)
  - Rappresentati come sequenze di caratteri, similmente alle stringhe in ASCII
- bytearray
  - Come il tipo bytes ma *mutabile*

N.B.: Questi tipi di dati sono stati implementati anche nella versione 2.7 ma non nelle versioni precendenti (nella 2.7 il tipo **bytes** è assimilato al tipo stringa **str**).

Il tipo None

- Il tipo None (**NoneType**) rappresenta l'oggetto nullo
- Usato in tutte le situazioni in cui si vuole indicare la mancanza di un valore definito
- Questo tipo assume solo un valore, ossia `None`:

```
>>> p = None
>>> p is None
True
>>> p == None     # Forma deprecata ... (PEP8)
True
>>> p
>>> print(p)
None
```

Booleani

- Il tipo **bool** è sostanzialmente un sottoinsieme degli interi (**int**)
- Un oggetto booleano può assumere solo valore **True** o valore **False**
- Esempi:

```
>>> a = True
>>> b = False
>>> print(a, b)
(True, False)
>>> int(a), int(b)
(1, 0)
```

- *Ma ogni oggetto in Python ha un valore booleano ...*

Tipi di dati strutturati

- Liste
- Tuple
- Insiemi
- Dizionari

Liste

- Una **lista** è una *sequenza ordinata mutabile* di elementi

```
l1 = []                     # Lista vuota
l2 = [10]                   # Lista con un solo elemento
l3 = [10, 20, 30]           # Lista con 3 valori interi
l4 = ['Ciao', 10, 20, 30]   # Lista con elementi vari
l5 = list('ABC')            # Equivale alla lista ['A', 'B', 'C']
```

- Accesso agli elementi:

```
>>> l4[0]
'Ciao'
>>> l4[-1]
30
>>> l4[0:-1]
('Ciao', 10, 20)
>>> l4[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Tuple

- Una **tupla** è una *sequenza ordinata immutabile* di elementi:

```
t1 = ()                     # Tupla vuota
t2 = (10,)                  # Tupla con un solo elemento (o anche t2 = 10,)
t3 = (10, 20, 30)           # Tupla con 3 valori interi
t4 = ('Ciao', 10, 20, 30)   # Tupla con elementi di tipo diverso
t5 = tuple('ABC')           # Crea la tupla ('A', 'B', 'C')
```

- Accesso agli elementi:

```
>>> t4[0]
'Ciao'
>>> t4[-1]
30
>>> t4[0:-1]          # Esempio di slicing!
('Ciao', 10, 20)
>>> t4[10]            # Errore di accesso!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

Insiemi

- Gli insiemi sono *collezioni non ordinate di elementi univoci (hashable)*
- Introdotti come tipi in Python 2.4 (prima erano nelle librerie)
- Differenziati in **set** e **frozenset**

```
s1 = set([10, 20, 'A', 'B'])    # oppure s1 = {10, 20, 'A', 'B'}
s2 = frozenset(['A', 'B', 'C'])
```

- I **frozenset** sono *immutabili* mentre i **set** possono essere modificati:

```
s1.remove(10)
s1.add(30)
```

- Sui set e frozenset possono essere effettuate operazioni tipiche degli insiemi:

```
>>> s2 | s1     # Esempio: unione di un set e un frozenset
frozenset(['A', 'C', 'B', 20, 10])
```

- Documentazione sugli insiemi:
  - https://docs.python.org/2/library/stdtypes.html#set-types-set-frozenset

Dizionari 1/2

- I dizionari sono *collezioni non ordinate di oggetti mappate attraverso chiavi univoche*
- Le chiavi del dizionario possono essere anche di tipo diverso ma devono essere *hashable*
- I dizionari sono *array associativi*, altrimenti noti come *mappe* o *tabelle hash*

```
d1 = {'x': 10, 'y': 20, 'z': 30}      # Dizionario con chiavi stringa
d2 = {1: 'Alpha', 2: 'Beta', 'x': 0}  # Misto di tipi diversi
d3 = {}                               # Dizionario vuoto
d1 = dict(x=10, y=20, z=30)
d2 = dict([[1, 'Alpha'], [2, 'Beta'], ['x', 0]])
d3 = dict()
```

- Un altro modo per creare un dizionario da una sequenza di chiavi:

```
>>> d4 = dict.fromkeys([1, 2, 3], 0)  # Funzione in oggetto funzione!
>>> d4
{1: 0, 2: 0, 3: 0}
```

Dizionari 2/2

- Accesso ad un elemento del dizionario:

```
>>> d2 = {1: 'Alpha', 2: 'Beta', 'x': 0}
>>> d2[1]
'Alpha'
>>> d2['x']
0
>>> d2[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
```

- I dizionari includono molte funzioni con la quale possono essere modificati o manipolati:

```
>>> d2.values()    # Ritorna l'elenco dei valori del dizionario
['Alpha', 'Beta', 0]
```

- Documentazione ufficiale sui dizionari:
  - https://docs.python.org/2/library/stdtypes.html#mapping-types-dict

Buffering e Viste

- Gestire buffering di oggetti in forma grezza (raw bytes) mediante API:
  - Elimina la necessità di duplicarlo
  - Rivista in Python 3 con nuove API (Py_buffer)
  - https://docs.python.org/2/c-api/buffer.html
- **memoryview**
  - Pensati per Python 3, è stato fatto un *backporting* in Python 2.7+
  - Vista a livello di bytes su oggetti che supportano API di buffering
  - https://docs.python.org/2/library/stdtypes.html#memoryview-type
- **dictview**
  - Viste dinamiche su dizionari (chiavi, valori o elementi)
  - Quando il dizionario cambia le modifiche si riflettono anche sulla vista
  - https://docs.python.org/2/library/stdtypes.html#dictionary-view-objects

Valore booleano di un oggetto

- In Python ogni oggetto può essere trattato come valore di verità, vero o falso
- Sono considerati con valore vero:
  - Numeri diversi da zero (*evitare i **float** però ...*)
  - Sequenze non vuote (stringhe, tuple, liste)
- Sono considerati con valore falso:
  - Il numero 0
  - Sequenze vuote
  - None
- Il valore booleano è determinato dall'attributo `__nonzero__`
- *Questa regola vale anche per il valore di ritorno delle funzioni ...*

Altri tipi di dati nelle librerie

- La libreria standard di Python definisce diversi tipi di dati strutturati specializzati
  - datetime
  - *collections*
  - *array*
  - queue
  - mutex
  - heapq
  - ...
- Sono in generale delle classi specializzate, alcune pensate per essere incrociate con altre classi definite dall'utente

Collezioni avanzate

- Nella libreria *collections* ci sono una serie di strutture dati avanzate:
  - **namedtuple** – Tuple con attributi (2.6+)
  - **deque** – Collezione tipo stack/coda (2.4+)
  - **Counter** – contatore di elementi unici (2.7+)
  - **OrderedDict** – Dizionario ordinato secondo l'inserimento (2.7+)
  - **defaultdict** – Dizionario con default per elementi mancanti (2.5+)
- Altre sono state aggiunte nelle versioni di Python 3:
  - https://docs.python.org/2.7/library/collections.html
  - https://docs.python.org/3.5/library/collections.html

Namedtuple

- Sono delle tuple in cui i vari elementi sono caratterizzati da un nome-attributo
- Usati per definire tipi con campi in modo stringato ed efficace, utili quando si vuole tenere un riferimento al nome dei campi (tuple di un Data Base ad esempio)
- Esempio:

```
>>> import collections
>>> Coord = collections.namedtuple('Coord3D', ['x', 'y', 'z'])
>>> p1 = Coord(1, 20, 0)
>>> p1
Coord3D(x=1, y=20, z=0)
>>> print(p1)
Coord3D(x=1, y=20, z=0)
>>> print(p1.x)
1
>>> print(p1.x, p1.y, p1.z)
(1, 20, 0)
>>> print(p1[0], p1[1], p1[2])
(1, 20, 0)
```

Array

- L'oggetto **array** permette di creare strutture del tutto analoghe alle liste ma ottimizzate per elementi numerici di tipo fisso
- Gli elementi di un array devono essere tutti dello stesso tipo:
  - https://docs.python.org/2/library/array.html
- Esempio:

```
>>> a = array.array('i', [10, 20, -1])
>>> print(a)
array('i', [10, 20, -1])
>>> print(a[0])
10
>>> a.append(100)
>>> print(a)
array('i', [10, 20, -1, 100])
>>> a.append(80.)

Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    a.append(80.)
TypeError: integer argument expected, got float
```

Assegnamento multiplo

- Assegnamento multiplo semplice:

```
>>> a = b = c = 0
>>> a, b, c
(0, 0, 0)
>>> a = 1
>>> a, b, c
(1, 0, 0)
>>> a = b = [1, 2, 3]    # Attenzione ai contenitori mutabili!
>>> a.append(4)
>>> a, b
([1, 2, 3, 4], [1, 2, 3, 4])
```

- Assegnamento multiplo parallelo (c'è anche in Go, Ruby, Perl, Javascript):

```
>>> a, b, c = 1, 2, 3
>>> a, b, c
(1, 2, 3)
>>> x, y = y, x   # Swap!
```

Assegnamento di oggetti immutabili

- In Python se assegno con un oggetto immutabile non c'è duplicazione a livello di memoria:

```
>>> a = (1, 2, 3, 4, 5,)
>>> b = a
>>> id(a) == id(b)    # in CPython id() è l'indirizzo in memoria
True
```

- La separazione a livello di memoria avviene dinamicamente se riassegno una delle variabili:

```
>>> a = (1, 2, 3, 4, 5,)
>>> b = a
>>> a = ('a', 'b', None, [])
>>> b    # La copia continua a mantenere il valore originale
(1, 2, 3, 4, 5)
```

- Inoltre tutti gli interi <= 256 condividono sempre la stessa zona di memoria:

```
>>> a = 10
>>> b = 10
>>> id(a) == id(b)
True
>>> a += 1
>>> id(a) == id(b)
False
>>> a -= 1
>>> id(a) == id(b)    # Tornano a condividere la stessa memoria
True
```

Assegnamento o modifica di oggetti *mutabili*

- Anche quando copio oggetti *mutabili* non viene effettuato un duplicato a livello di memoria:

```
>>> elenco1 = [10, 15, 20]
>>> elenco2 = elenco1
>>> id(elenco1) == id(elenco2)
True
>>> hex(id(elenco1)), hex(id(elenco2))
('0x7f53e14bf950', '0x7f53e14bf950')
>>> elenco1.append(25)
>>> hex(id(elenco1)), hex(id(elenco2))
('0x7f53e14bf950', '0x7f53e14bf950')
>>> elenco2     # Le modifiche si riflettono su entrambi gli oggetti
[10, 15, 20, 25]
>>> elenco1 = ['a', 'b', 'c']
>>> elenco2     # Se invece riassegno uno l'altro non viene alterato
[10, 15, 20, 25]
```

- Per duplicare la memoria si può usare il *costruttore* di lista (funzione built-in *list()*):

```
>>> elenco1 = [10, 15, 20]
>>> elenco2 = list(elenco1)
>>> id(elenco1) == id(elenco2)
False
```

Copia di oggetti mutabili

- Per replicare strutture dati mutabili c'è la libreria **copy**
  - https://docs.python.org/2/library/copy.html
- Due tipi di replicazione:

  `copy.copy(x)`: Copia superficiale (un livello)

  `copy.deepcopy(x)`: Copia completa in profondità

- Esempio:

```
>>> import copy
>>> d1 = { 1: 'uno', 2: 'due', 3: 'tre' }
>>> d2 = copy.copy(d1)
>>> hex(id(d1)), hex(id(d2))  # Non è più la stessa memoria
('0x7f383d664168', '0x7f383d664a28')
>>> d1[4] = 'quattro'    # Se modifico d1 l'altro rimane uguale
>>> d1
{1: 'uno', 2: 'due', 3: 'tre', 4: 'quattro'}
>>> d2
{1: 'uno', 2: 'due', 3: 'tre'})
```