

Elementi del linguaggio introdotti

Sino ad ora in Python abbiamo visto:

1. Dati numerici (3, 4, 1.41)
2. Operatori aritmetici, in notazione prefix (`add`, `sub`, `mul`, `truediv`) o post-fix (+, -, *, /)
3. Operatori di confronto (>, >=, <, <=, ==)
4. Operatori logici (**and**, **or**, **not**)
5. La definizione di procedure, eventualmente ricorsive, tramite le parole chiavi **def** e **return**
6. Espressioni condizionali:
 if <predicato>:
 <block1>
 else:
 <block2>
7. La procedura `print()` e il comando `who`

Concetti di Programmazione


1. Valutazione di espressioni semplici e composte tramite il ciclo *read-evaluate-print*
 - a) Metodo di sostituzione: prima valuta gli argomenti e poi sostituisci, chiamato “*applicative-order evaluation*”
 - b) Prima sostituisci le procedure e poi riduci, chiamato “*normal-order evaluation*”
2. Ricerca di un valore per (e.g., radice quadrata di x):
 - a) Enumerazione esaustiva
 - b) Per bisezione
 - c) Per approssimazioni successive (e.g., Newton)
3. Processi di calcolo di tipo:
 - a) Processo ricorsivo lineare (e.g. Fattoriale)
 - b) Processo iterativo lineare (e.g. Fattoriale)
 - c) Processo con ricorsione ad albero (e.g. Fibonacci)

Errori Comuni 1/2

- Errori di sintassi:

```
In [1]: if a == b
        print("ciao")


File "<ipython-input-1-bc040a8b9a79>", line 1
      if a == b
              ^
SyntaxError: invalid syntax
```



- Errori di mancata definizione di variabili:

```
In [4]: if a == b:
        print("ciao")

-----
NameError                                Traceback (most recent call last)
<ipython-input-4-c47d6e7b2bfe> in <module>()
----> 1 if a == b:
      2     print("ciao")
```




NameError: name 'a' is not defined


Errori Comuni 2/2

- Errori di chiamate ricorsive senza terminazione:

```
In [7]: def P():  
        return P()  
        P()
```




```
-----  
RecursionError                                Traceback (most recent call last)  
<ipython-input-7-a392e54bf72a> in <module>()  
      1 def P():  
      2     return P()  
----> 3 P()  
  
<ipython-input-7-a392e54bf72a> in P()  
      1 def P():  
----> 2     return P()  
      3 P()
```



```
... last 1 frames repeated, from the frame below ...
```

```
<ipython-input-7-a392e54bf72a> in P()  
      1 def P():  
----> 2     return P()  
      3 P()
```

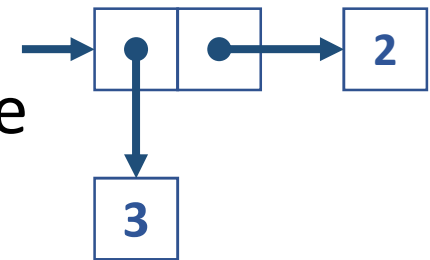


```
RecursionError: maximum recursion depth exceeded
```

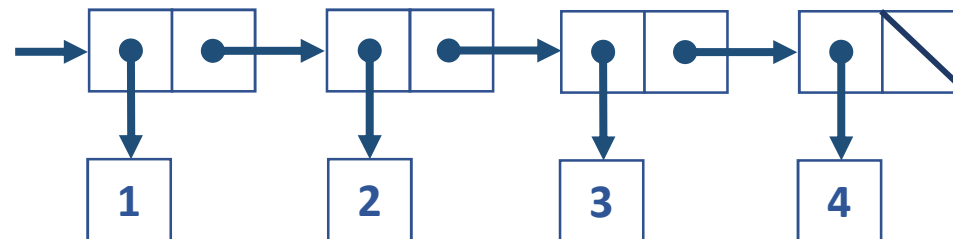
Strutture dati composte

Abbiamo definito dati composti usando gli elementi base del linguaggio con due esempi fondamentali:

1. I **numeri razionali**, definiti da numeratore e denominatori (esempio: 3/2)



2. Le **pairslist**, definiti come catena di coppie



Entrambi i tipi di dati sono definiti a partire dalle **coppie (pairs)** che sono in pratica **tuple** di lunghezza due.

Numeri razionali

Funzioni per creare un nuovo oggetto di “tipo” numero razionale, ovvero il **COSTRUTTORE**:

```
def MakeQ(n, d) :  
    mcd = MCD(n, d)  
    return (n/mcd, d/mcd)
```

Funzioni per accedere ai dati base del dato composto numero razionale, ovvero i **SELETTORI**:

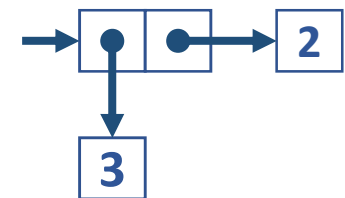
```
def Num(x) :  
    return x[0]
```

```
def Den(x) :  
    return x[1]
```

Application Programming Interface (API)

Struttura dati per gestire una coppia di numeri

Abbiamo usato le coppie di Python



Application Programming Interface (API)

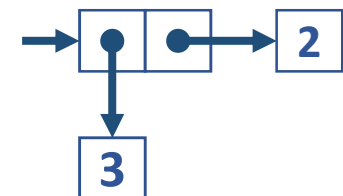
Struttura dati “Num. Razionale”: MakeQ, Num, Den

Definizione **COSTRUTTORE E SELETTORI**

Struttura dati per gestire una coppia di numeri

Abbiamo usato le coppie di Python

```
def MakeQ (n, d) :  
def Num (x) :  
def Den (x) :
```



Funzioni su Numeri razionali

Abbiamo definito le funzioni più comuni per utilizzare i dati composti di tipo “numero razionale”

Funzioni di supporto alle operazioni aritmetiche:

```
def AddQ(x, y):  
    return MakeQ(Num(x)*Den(y)+Num(y)*Den(x), Den(x)*Den(y))
```

```
def SubQ(x, y):  
    return MakeQ(Num(x)*Den(y)-Num(y)*Den(x), Den(x)*Den(y))
```

```
def MulQ(x, y):  
    return MakeQ(Num(x)*Num(y), Den(x)*Den(y))
```

```
def DivQ(x, y):  
    return MakeQ(Num(x)*Den(y), Den(x)*Num(y))
```

Predicato di confronto:

```
def EqualQ(x, y):  
    return Num(x)*Den(y) == Num(y)*Den(x)
```

Application Programming Interface (API)

Definizione delle procedure AddQ, SubQ, EqualQ,...

Definizione Funzioni aritmetiche per i numeri razionali

```
def Equal(x, y):  
def AddQ(x):  
def SubQ(x):  
...
```

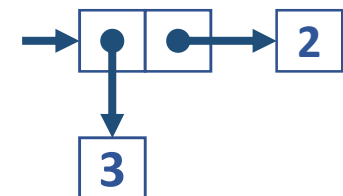
Struttura dati "Num. Razionale": MakeQ, Num, Den

Definizione **COSTRUTTORE E SELETTORI**

```
def MakeQ(n, d):  
def Num(x):  
def Den(x):
```

Struttura dati per gestire una coppia di numeri

Abbiamo usato le coppie di Python



Application Programming Interface (API)

Programmi che usano i numeri razionali

Numeri razionali usati in una certa applicazione

Definizione delle procedure AddQ, SubQ, EqualQ,...

Definizione Funzioni aritmetiche per i numeri razionali

```
def Equal(x, y):  
def AddQ(x):  
def SubQ(x):  
...
```

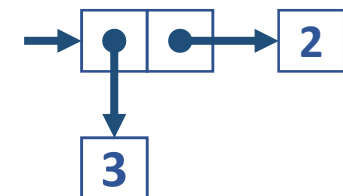
Struttura dati "Num. Razionale": MakeQ, Num, Den

Definizione **COSTRUTTORE E SELETTORI**

```
def MakeQ(n, d):  
def Num(x):  
def Den(x):
```

Struttura dati per gestire una coppia di numeri

Abbiamo usato le coppie di Python



Pairslist

Funzioni per creare un nuovo oggetto di “tipo” **pairslist**, ovvero i **COSTRUTTORI**:

```
def EmptyList():  
    return None
```

```
def MakeList(x, y=None):  
    return (x, y)
```

```
def MakeRange(a, b):  
    def MakeI(n):  
        if n > b:  
            return EmptyList()  
        return MakeList(n, MakeI(n+1))  
    return MakeI(a)
```

Funzioni per accedere ai dati base del dato composto **pairslist**, ovvero i **SELETTORI**:

```
def Head(As):  
    return As[0]
```

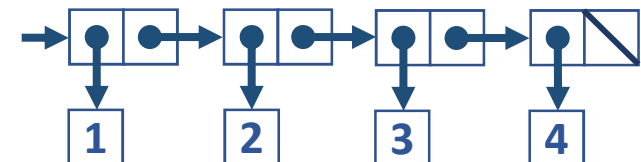
```
def Tail(As):  
    return As[1]
```

```
def Nth(As, i):  
    if IsEmpty(As):  
        return As  
    if i == 0:  
        return Head(As)  
    return Nth(Tail(As), i-1)
```

Application Programming Interface (API)

Struttura dati per gestire le pairslits

Abbiamo usato catene di coppie



Application Programming Interface (API)

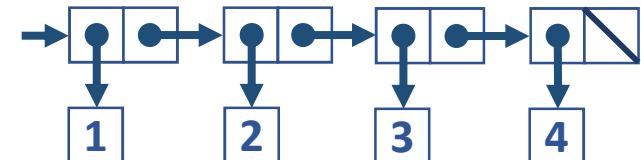
Supporto di base per struttura dati "PairsList"

Definizione **COSTRUTTORE E SELETTORI**

Struttura dati per gestire le pairslits

Abbiamo usato catene di coppie

```
def MakeList  
def Head(As) :  
def Tail(As) :
```



Funzioni su Pairslist

Abbiamo un insieme di funzioni che ci permettono di utilizzare il dato compost “pairslist”, che generalizza il concetto di “sequenza di elementi”

Funzione	Descrizione
Equal (As, Bs)	Confronto elemento per element
Length (As)	Lunghezza della lista
Append (As, Bs)	Concatena Bs a As (operatore ‘+’ ?)
Contains (As, value)	Controlla se “value” è un elemento della lista
RemoveFirst (As, value)	Rimuovi il primo elemento uguale a “value”
RemoveAll (As, value)	Rimuovi tutti gli elementi uguale a “value”
Count (As, value)	Conta il numero di volte che “value” appare in As
Min (As)	Trova il minimo in As
Max (As)	Trova il massimo in As
Sum (As)	Calcola la somma di tutti gli elementi in As (somma definita?)
Prod (As)	Calcola la produttoria di tutti gli elementi in As (prod definito?)
Reverse (As)	Inverti la sequenza degli elementi nella lista As
MakeRandomInts (n, a, b)	Crea una lista di “n” numeri pseudo-casuali interi in [a,b] uniforme

Application Programming Interface (API)

Definizione di funzioni per “usare” le pairlist

Funzioni più comuni per sequenze di oggetti

```
def Equal(x, y) :  
def Append(x) :  
def Length(x) :  
...
```

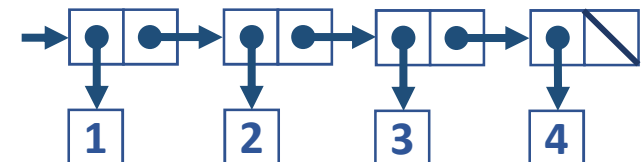
Supporto di base per struttura dati “PairsList”

Definizione **COSTRUTTORE E SELETTORI**

```
def MakeList  
def Head(As) :  
def Tail(As) :
```

Struttura dati per gestire le pairlists

Abbiamo usato catene di coppie



Funzioni su Pairslist: pattern ricorrenti

Nel definire le funzioni di base sulle **pairslist**, si possono notare degli **schemi ricorrenti** ("**pattern**") che ci permettono di definire tre procedure molto generali che operano su sequenze di oggetti:

```
def Map(F, As):          # "F" è una funzione
    if IsEmpty(As):
        return As
    return MakeList(F(Head(As)), Map(F, Tail(As)))
```

```
def Filter(P, As):       # "P" è un predicato
    if IsEmpty(As):
        return As
    if P(Head(As)):
        return MakeList(Head(As), Filter(P, Tail(As)))
    return Filter(P, Tail(As))
```

Chiamata anche **Reduce**

```
def FoldRight(Op, As, z): # "Op" è una funzione che prende due input
    if IsEmpty(As):
        return z
    return Op(Head(As), FoldRight(Op, Tail(As), z))
```

Application Programming Interface (API)

Funzioni che individuano Pattern per le pairslist

Funzioni molto importanti

```
def Map(x, y) :  
def Filter(x) :  
def Fold(x) :  
...
```

Definizione di funzioni per “usare” le pairslist

Funzioni più comuni per sequenze di oggetti

```
def Equal(x, y) :  
def Append(x) :  
def Length(x) :  
...
```

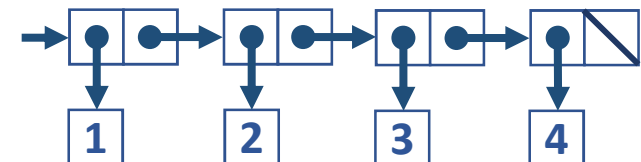
Supporto di base per struttura dati “PairsList”

Definizione **COSTRUTTORE E SELETTORI**

```
def MakeList  
def Head(As) :  
def Tail(As) :
```

Struttura dati per gestire le pairslists

Abbiamo usato catene di coppie



Application Programming Interface (API)

Programmi che usano le pairlist

Pairslist usate in programmi di “alto livello”

Funzioni che individuano Pattern per le pairlist

Funzioni molto importanti

```
def Map(x, y) :  
def Filter(x) :  
def Fold(x) :
```

...

Definizione di funzioni per “usare” le pairlist

Funzioni più comuni per sequenze di oggetti

```
def Equal(x, y) :  
def Append(x) :  
def Length(x) :
```

...

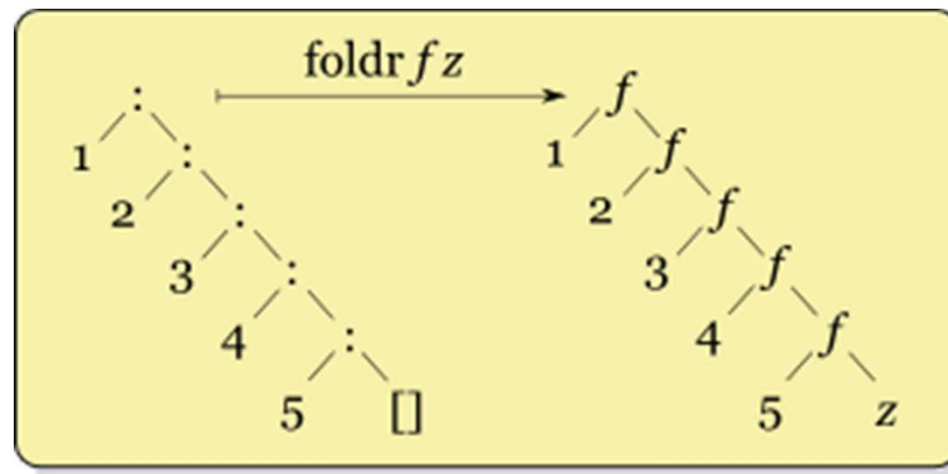
Supporto di base per struttura dati “PairsList”

Definizione **COSTRUTTORE E SELETTORI**

```
def MakeList  
def Head(As) :  
def Tail(As) :  
def EmptyList() :
```

Funzioni su Pairslist: pattern ricorrenti

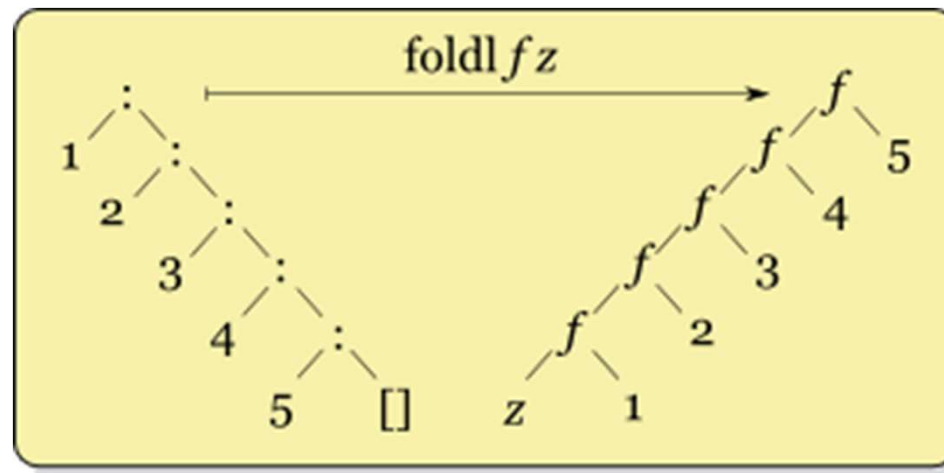
Abbiamo notato come molte delle funzioni della tabella precedente possano essere definite in termini della funzione “Fold”, e che la stessa può essere definita in due modi: **FoldRight** e **FoldLeft**



$(1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + (9 + (10 + (11 + (12 + (13 + 0))))))))))))))$

Funzioni su Pairslist: pattern ricorrenti

Abbiamo notato come molte delle funzioni della tabella precedente possano essere definite in termini della funzione “Fold”, e che la stessa può essere definita in due modi: **FoldRight** e **FoldLeft**



`(((((0+1)+2)+3)+4)+5)+6)+7)+8)+9)+10)+11)+12)+13)`

Dati composti *builtin* in Python

La possibilità di manipolare sequenze di elementi è un elemento di programmazione che in Python esiste una struttura dati:

`list()`

implementata direttamente nel linguaggio.

Oltre alle liste, abbiamo visto altri dati composti builtin:

1. `tuple()` , ovvero sequenze di oggetti (non modificabili)
2. `str()` , ovvero sequenze di caratteri (non modificabili)
3. `dict()` , ovvero insieme di coppie di dati (*key, value*), indicizzati dalla chiave

Ci mancano da vedere

1. `set()` , ovvero insieme di oggetti “unici”
2. `range()` , ovvero sequenze implicite di interi

Funzioni *builtin* in Python

Oltre ai dati composti, in Python ci sono una serie di funzioni, predicati, e procedure che sono **builtin** del linguaggio:

Vedere <https://docs.python.org/3/library/functions.html>

2. Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are lis

Built-in Functions				
<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

2. Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are lis

Built-in Functions				
<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

2. Built-in Functions

- Funzioni già viste e usate
- Funzioni da vedere a breve

The Python interpreter has a number of functions and types built into it that are always available. They are listed below.

Built-in Functions					
abs() ●	dict() ●	help()	min() ●	setattr()	
all() ●	dir()	hex()	next() ●	slice()	
any() ●	divmod()	id() ●	object()	sorted() ●	
ascii()	enumerate()	input() ●	oct()	staticmethod()	
bin()	eval()	int() ●	open() ●	str() ●	
bool() ●	exec()	isinstance()	ord() ●	sum() ●	
bytearray()	filter() ●	issubclass()	pow() ●	super()	
bytes()	float() ●	iter()	print() ●	tuple() ●	
callable()	format()	len() ●	property()	type()	
chr() ●	frozenset()	list() ●	range() ●	vars()	
classmethod()	getattr()	locals()	repr()	zip() ●	
compile()	globals()	map() ●	reversed() ●	__import__()	
complex() ●	hasattr()	max() ●	round() ●		
delattr()	hash()	memoryview()	set() ●		

Confronto *semantico* Pairslist e list()

Pairslist	List
Equal (As, Bs)	"=="
Length (As)	len (As)
Append (As, Bs)	As.append() oppure "As + Bs"
Contains (As, value)	"value in As"
RemoveFirst (As, value)	As.remove (value)
RemoveAll (As, value)	Esercizio, con filter
Count (As, value)	As.count (value)
Min (As)	min (As)
Max (As)	max (As)
Sum (As)	sum (As)
Prod (As)	In termini di reduce
Reverse (As)	reversed (As)
MakeRandomInts (n, a, b)	Maniera analoga, user defined