In Python everything is an object

- Basic types are objects, e.g.
  - lists, integers, float, bool
- Object methods
- range is a nice simple object
- Lists revisited

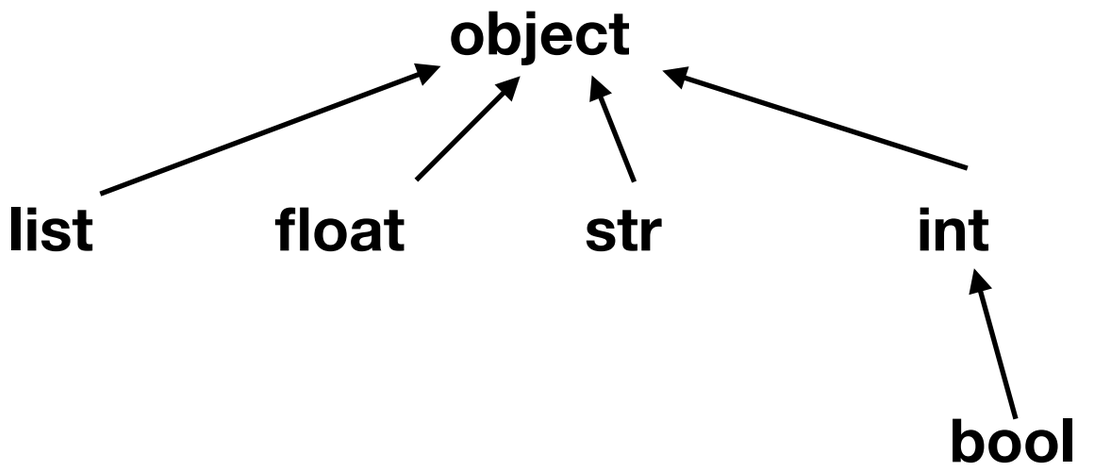
In Python everything is an object

```
x = 3
y = 3.14
z = "Hello"
w = [3,4,5]
b = False
In [46]:
print(x, type(x))
print(y, type(y))
print(z, type(z))
print(w, type(w))
print(b, type(b))

3 <class 'int'>
3.14 <class 'float'>
Hello <class 'str'>
[3, 4, 5] <class 'list'>
False <class 'bool'>
```

```
help(w)

Help on list obj

class list(objec
 |  list() -> ne
…
```

```
help(x)

Help on int object:

class int(object)
 |  int(x=0) -> integer
…
```

```
help(b)

Help on bool object:

class bool(int)
 |  bool(x) -> bool
…
```

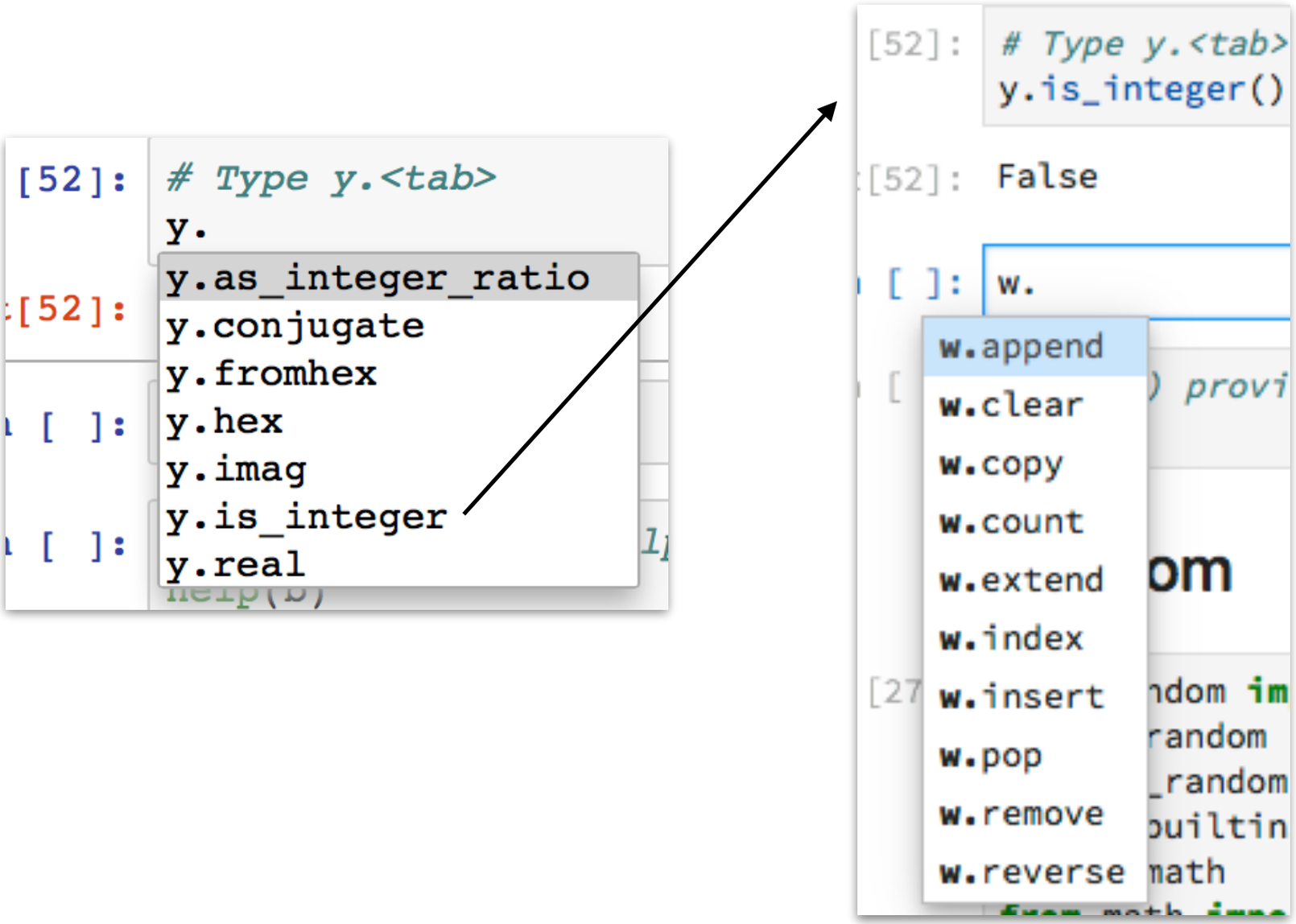


What does this mean?

- In C or Java an int is stored in 2 or 4 bytes (16 or 36 bits)
- In Python ints are fully fledged objects with lots of paraphernalia
  - `sys.getsizeof(x)` returns 28 bytes!
    - you need to **`import`** `sys` to run this

Objects have Methods

- <tab> completion in Jupyter Notebooks
  - y.<tab> shows list of methods/attributes for y



range is a nice simple class in Python

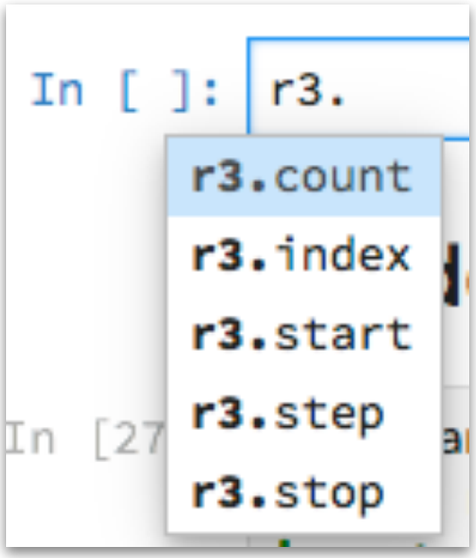
```
el = [3,6,9]
for i in el: print(i)

print("----")
for i in range(7):print(i)

3
6
9
----
0
1
2
3
4
5
6
```

- Python has a beautiful loop syntax
  - for <iter_var> in <sequence>
- But sometimes you want access to the loop index

```
r1 = range(7)
help(r1)

Help on range object:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
```

range class

- Immutable sequence type
- Five tab completions

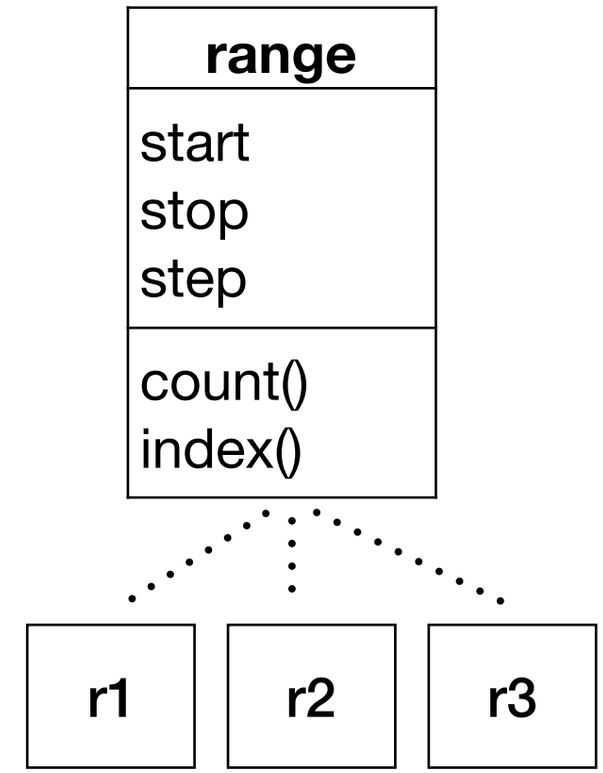


- Two methods -
  - count() & index()
- Three attributes
  - start, stop & step

```
r3.step
5


r3.index(10)
2
```

```
r1 = range(7)
r2 = range(2,7)
r3 = range(0,30,5)
In [8]:
for i in r1: print(i)
print('----')
for i in r2: print(i)
print('----')
for i in r3: print(i)
```

```
0
1
2
3
4
5
6
----
2
3
4
5
6
----
0
5
10
15
20
25
```



Lists (again as objects)

- Lets look again at lists
  - focusing on the fact that they are implemented as objects
- Traversing a list
  - using indices
- Updating a list
- Mapping, Filtering, Reducing lists
- List Methods in Python
- Using lists as arrays in Python

Traversing a list

- for … in syntax is real elegant

```
for cheese in cheeses:
    print(cheese)

Cheddar
Edam
Gouda
```

- You can also introduce an index…

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
numbers
Out[11]:
[136, 616]
```

Aside: The Pythonic Way

- Python aficionados maintain there is one right way to do things
  - and using `range(len(cheeses)` is not great:

```
for i in range(len(cheeses)):
    print(i,cheeses[i])

0 Cheddar
1 Edam
2 Gouda
```

  - use `for i, cheese in enumerate(cheeses):`

```
# The Pythonic way
cheeses[1] = 'Gubbeen'
for i, cheese in enumerate(cheeses):
    print(i,cheese)

0 Cheddar
1 Gubbeen
2 Gouda
```

Aside: Aside: Enumerate

- enumerate adds an index to an ‘iterable’
- creates an enumerate object
  - index/item tuples

```
list (enumerate(cheeses))
Out[11]:
[(0, 'Cheddar'), (1, 'Gubbeen'), (2, 'Gouda')]
```

- iterable: something that can be iterated over
  - typically a sequence, set, list

Updating a list

- Remember, lists are mutable (unlike strings)
  - So we can change elements

```
numbers = [17, 123]
numbers
Out[2]:
[17, 123]

numbers[1] = 5      # Lists are mutable
numbers
Out[5]:
[17, 5]
```

- Two simple ways to extend lists
  - using `.append` method
  - the + operator
    - 'hidden' method `__add__` defines behaviour for '+'

```
t = [33,44,55,66]
x = 99
t.append(x)
t
Out[31]:
[33, 44, 55, 66, 99]

t = [33,44,55,66]
x = 99
t = t + [x]
t
Out[32]:
[33, 44, 55, 66, 99]
```

4 ways not to update a list…

- Can you predict what happens in the following scenarios
  - Use a Notebook to test.

```
t.append([x])
t = t.append(x)
t + [x]
t = t + x
```

1

```
t = [33,44,55,66]
x = 99
t.append([x])
```

2

```
t = [33,44,55,66]
x = 99
t = t.append([x])
```

3

```
t = [33,44,55,66]
x = 99
t + [x]
```

4

```
t = [33,44,55,66]
x = 99
t = t + x
```

4 ways not to update a list…

1

```
t = [33,44,55,66]
x = 99
t.append([x]) # Wrong, produces a nested list
t
```

```
Out[1]:
[33, 44, 55, 66, [99]]
```

2

```
t = [33,44,55,66]
x = 99
t = t.append([x]) # Wrong, returns nothing
t
```

3

```
t = [33,44,55,66]
x = 99
t + [x]           # Wrong, doesn't change t
t
```

```
Out[3]:
[33, 44, 55, 66]
```

4

```
t = [33,44,55,66]
x = 99
t = t + x         # Wrong, adding an int to a list
```

Map, Filter, Reduce

Categories of operations on lists:

- Reduce:
  - An operation across all list members,
  - produces a single result.
- Map:
  - Same operation on all list members,
  - produces a list of results.
- Filter:
  - Filter function applied to all list members,
  - produces a smaller list - some members filtered out
- Other:
  - Lots of other stuff

Reduce

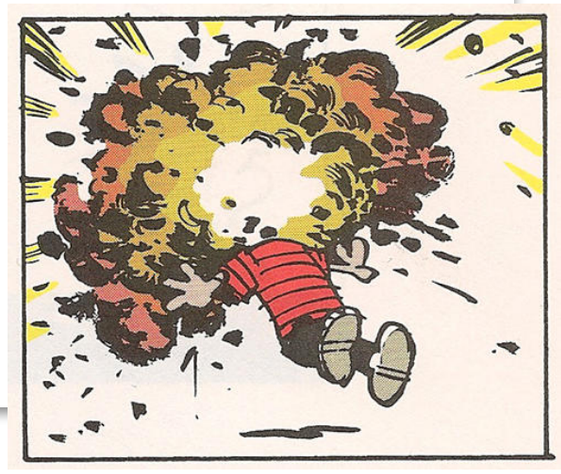
- An operation across all list members,
- produces a single result.

```
t = [33,44,55,66]
sum(t)
Out[6]:
198
```

Map

- Same operation on all list members,
- produces a list of results.

```
# Map: apply (map) a function to all elements in the list.
def square(e):
    return e*e

def myMapper(ls, funct):
    r =[]
    for e in ls:
        r.append(funct(e))
    return r

t = [33,44,55,66]
myMapper(t,square)
Out[17]:
[1089, 1936, 3025, 4356]
```



mind = blown

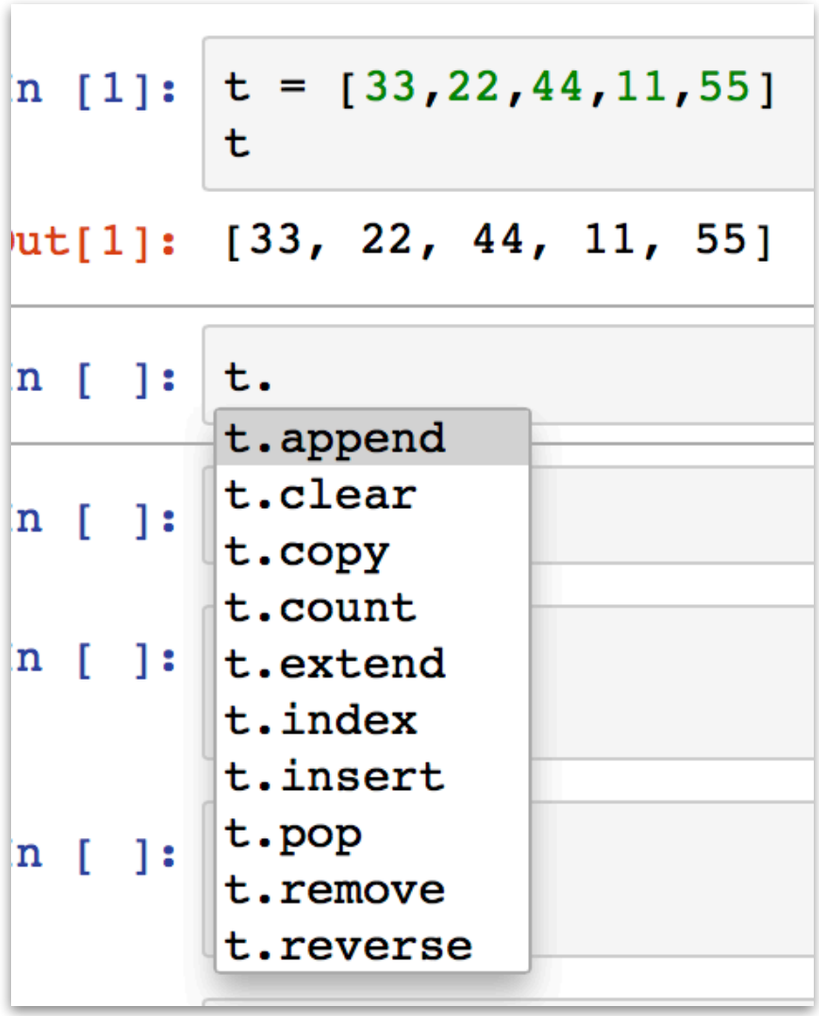
Filter

- Filter function applied to all list members,
- produces a smaller list - some members filtered out
-

```
# Filter: remove elements from a list based on a test.
def evenTest(e):
    if e % 2 == 0:
        return True
    return False

# filter function is passed in as argument
def myFilter(ls,filter):
    r =[]
    for e in ls:
        if filter(e): r.append(e)
    return r

myFilter(t,evenTest)
Out[15]:
[44, 66]
```

List Methods

- list.append(obj)
  - Appends object obj to list
- list.count(obj)
  - Returns count of how many times obj occurs in list
- list.extend(seq)
  - Appends the contents of seq to list
- list.index(obj)
  - Returns the lowest index in list that obj appears
- list.insert(index, obj)
  - Inserts object obj into list at offset index
- list.pop(obj=list[-1])
  - Removes and returns last object or obj from list
- sort([func])
  - Sorts objects of list, uses fund for comparison if supplied.
- list.remove(obj)
  - Removes object obj from list
- list.reverse
  - Reverses objects of list in place



List Methods - examples

- index
- sort
- pop
- remove

```
t = [33,22,44,11,55]
t
Out[27]:
[33, 22, 44, 11, 55]
```

```
t.index(11)
Out[28]:
3
```

```
t.sort()
t
Out[29]:
[11, 22, 33, 44, 55]
```

```
t.pop()
t
Out[30]:
[11, 22, 33, 44]
```

```
t.remove(33)
t
Out[31]:
[11, 22, 44]
```

In Python Everything is an Object

- Thinking of ints and lists as objects

Next

- The OO Creed
  - Encapsulation
  - Inheritance
  - Polymorphism
  - …
- Creating new Objects/Classes