Overloading & Special Methods

- **Method Overloading**
  - A class can have more than one **method** having the same name
  - if their argument lists are different.
  - Not possible in Python because it is ***dynamically typed***
- **Special Methods**
  - Dunders - double underscores
  - `__init__`
  - `__str__`
  - `__lt__`
  - `__len__`
  - `__add__`
- **Lambda Functions**

Constructor - Overloading

- What is wrong with this code?

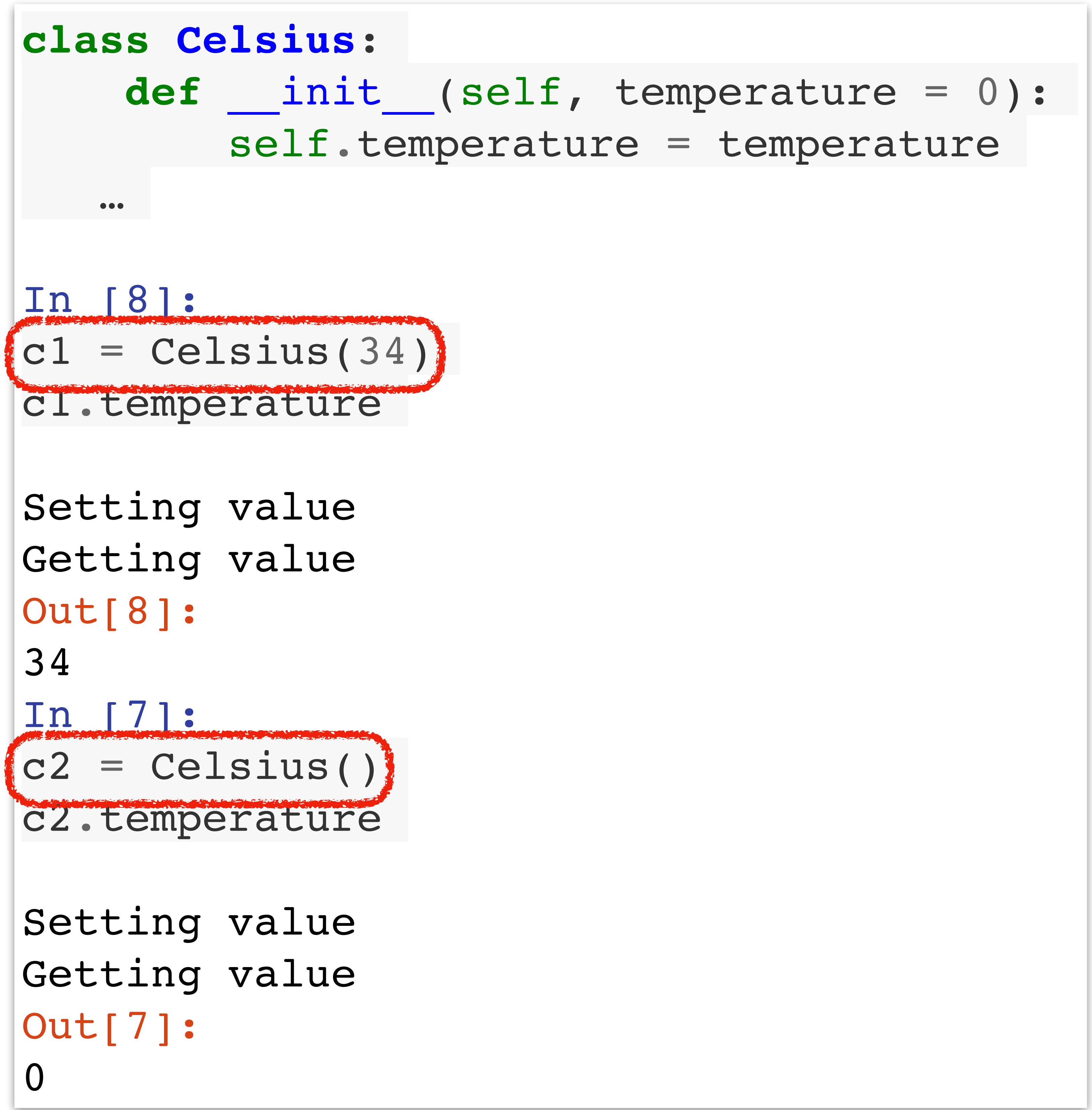
```
class StudentData
{
   private int stuID;
   private String stuName;
   private int stuAge;
   StudentData()
   {
       //Default constructor
       stuID = 100;
       stuName = "New Student";
       stuAge = 18;
   }
   StudentData(int num1, String str, int num2)
   {
       //Parameterized constructor
       stuID = num1;
       stuName = str;
       stuAge = num2;
   }
```

Nothing really
It’s just Java

Difference between Java and Python

Constructor - Overloading

- Multiple methods with same name
  - different arguments
- Not possible in Python
  - but we don't need it



```
class Celsius:
    def __init__(self, temperature = 0):
        self.temperature = temperature
  ...

In [8]:
c1 = Celsius(34)
c1.temperature

Setting value
Getting value
Out[8]:
34
In [7]:
c2 = Celsius()
c2.temperature

Setting value
Getting value
Out[7]:
0
```

Polymorphism - Overloading

- Add logic to the constructor to handle different argument possibilities.

```python
class Celsius:
    def __init__(self, s = None):
        if s is None:
            t = 0
        elif type(s) is str:
            t = int(s)
        else:
            t = s
        self.temperature = t
  …
```

```
t3 = Celsius(3)
t3.temperature
Out[15]:
3
In [16]:
t4 = Celsius('4')
t4.temperature
Out[16]:
4
In [17]:
t5 = Celsius()
t5.temperature
Out[17]:
0
```

Static vs Dynamic Typing

- Java is Statically Typed
  - Variable type is declared at compile time

```
class StudentData
{
    private int stuID;
    private String stuName;
    private int stuAge;
```

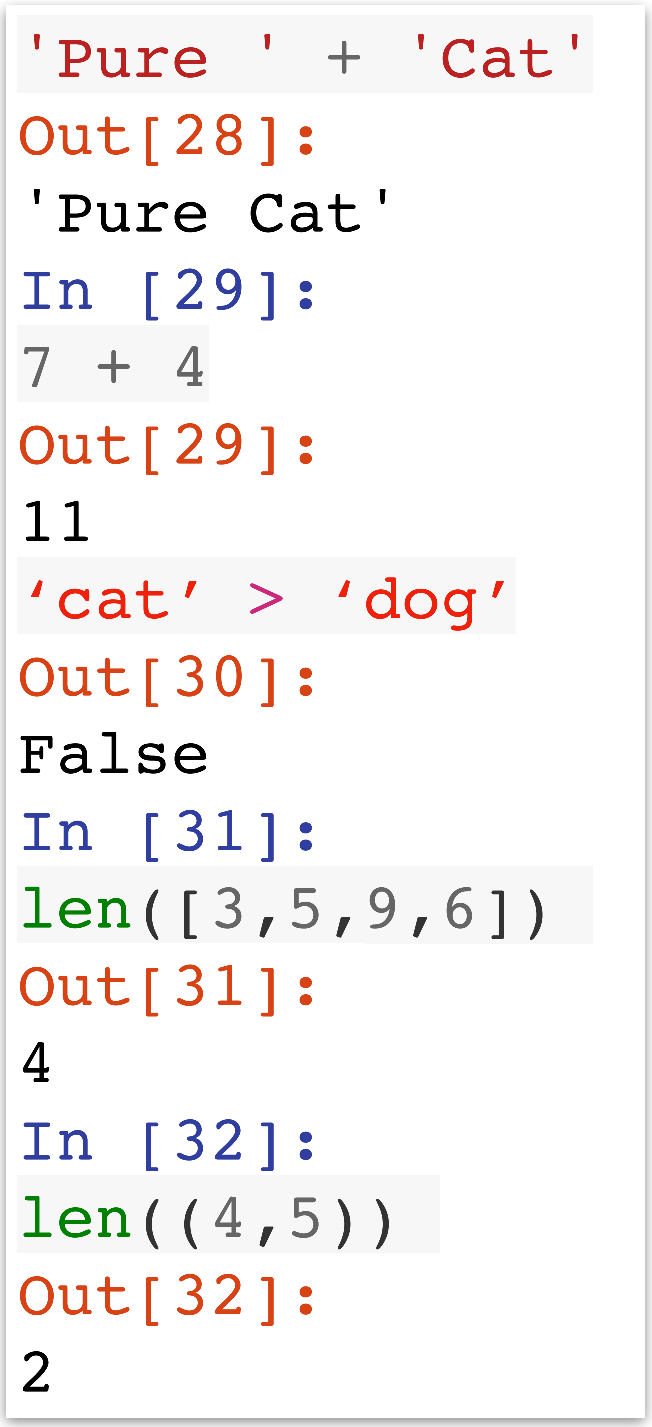
  - Type of all variables is known
  - Compiler can select correct version of method based on arguments
- Python is a Dynamically Typed Language
  - Type of a variable is not known at compile time

```
var = 7
var = 'seven'
var
```

  - This means method overloading is not possible

Special Methods

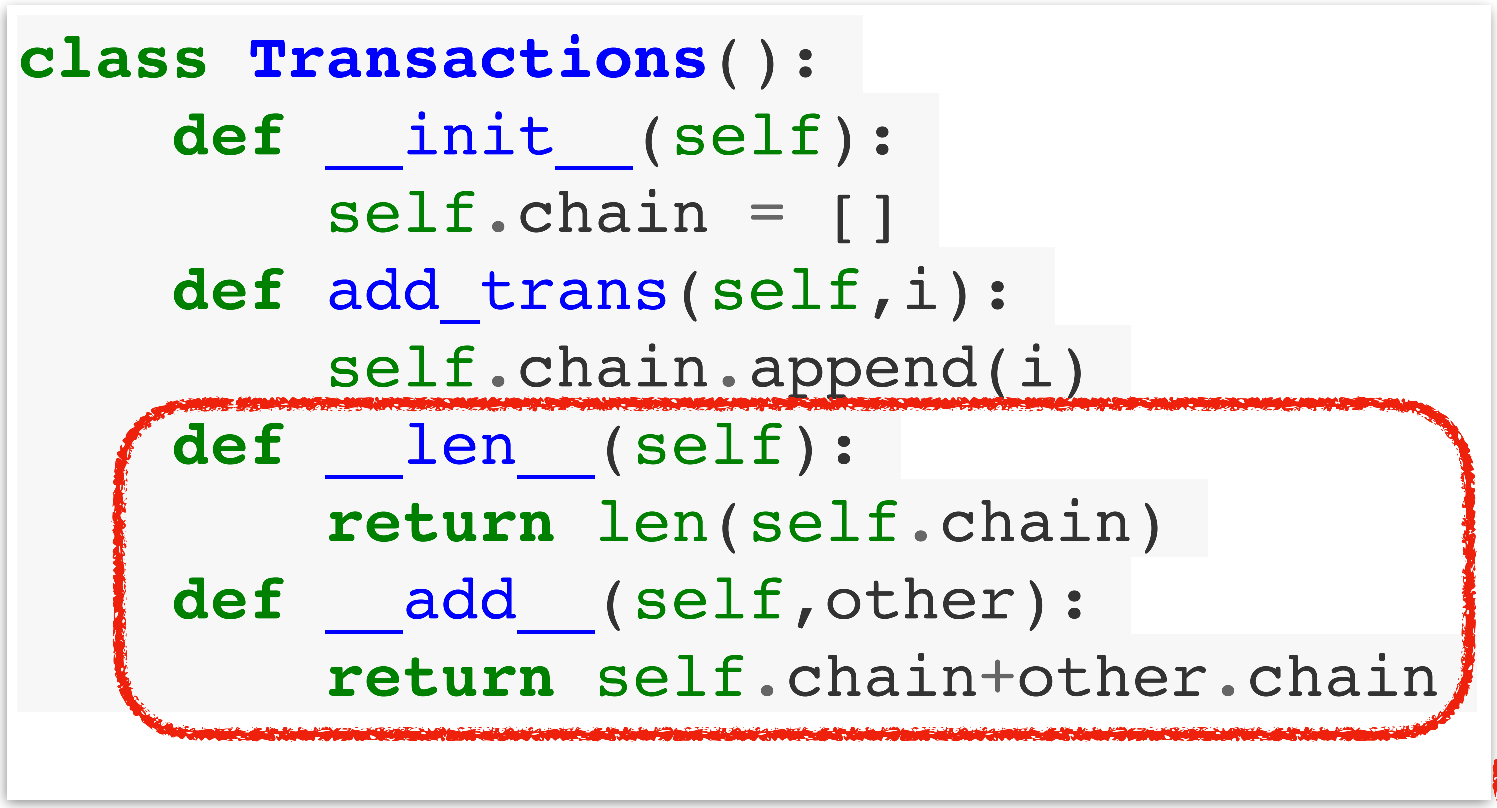
- Everything in Python is an object
- Some operators and functions are generic
  - e.g len or +
- Python provides hooks to implement these
  - __len__
  - __add__
  - __gt__
- Other Special Methods
  - __init__
  - __lt__
  - __str__



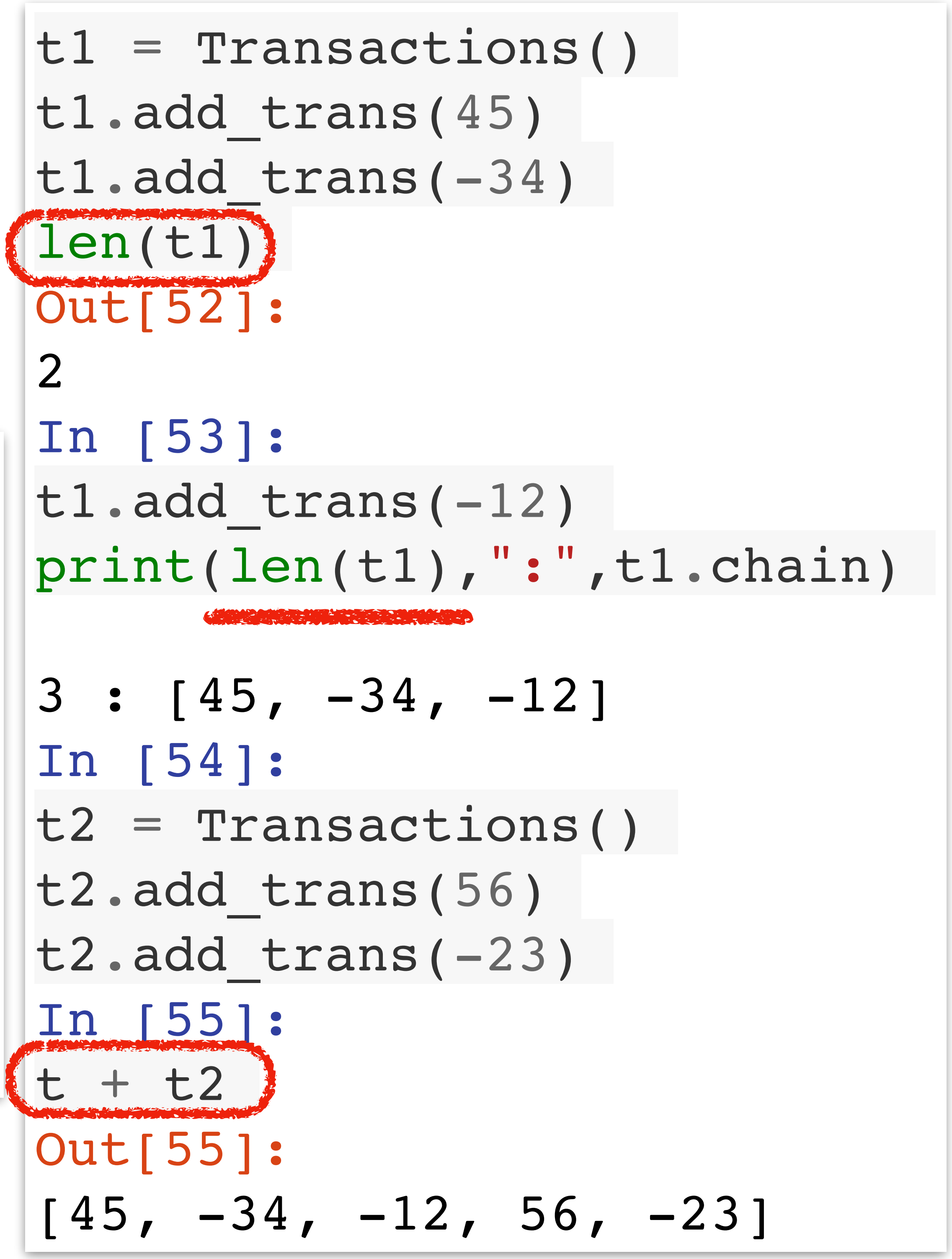
```
'Pure ' + 'Cat'
Out[28]:
'Pure Cat'
In [29]:
7 + 4
Out[29]:
11
'cat' > 'dog'
Out[30]:
False
In [31]:
len([3,5,9,6])
Out[31]:
4
In [32]:
len((4,5))
Out[32]:
2
```

Example with len and +

- A class which stores a sequence of transactions
  - define len and + behaviour



```python
class Transactions():
    def __init__(self):
        self.chain = []
    def add_trans(self,i):
        self.chain.append(i)
    def __len__(self):
        return len(self.chain)
    def __add__(self,other):
        return self.chain+other.chain
```



```
t1 = Transactions()
t1.add_trans(45)
t1.add_trans(-34)
len(t1)
Out[52]:
2
In [53]:
t1.add_trans(-12)
print(len(t1),":",t1.chain)

3 : [45, -34, -12]
In [54]:
t2 = Transactions()
t2.add_trans(56)
t2.add_trans(-23)
In [55]:
t + t2
Out[55]:
[45, -34, -12, 56, -23]
```

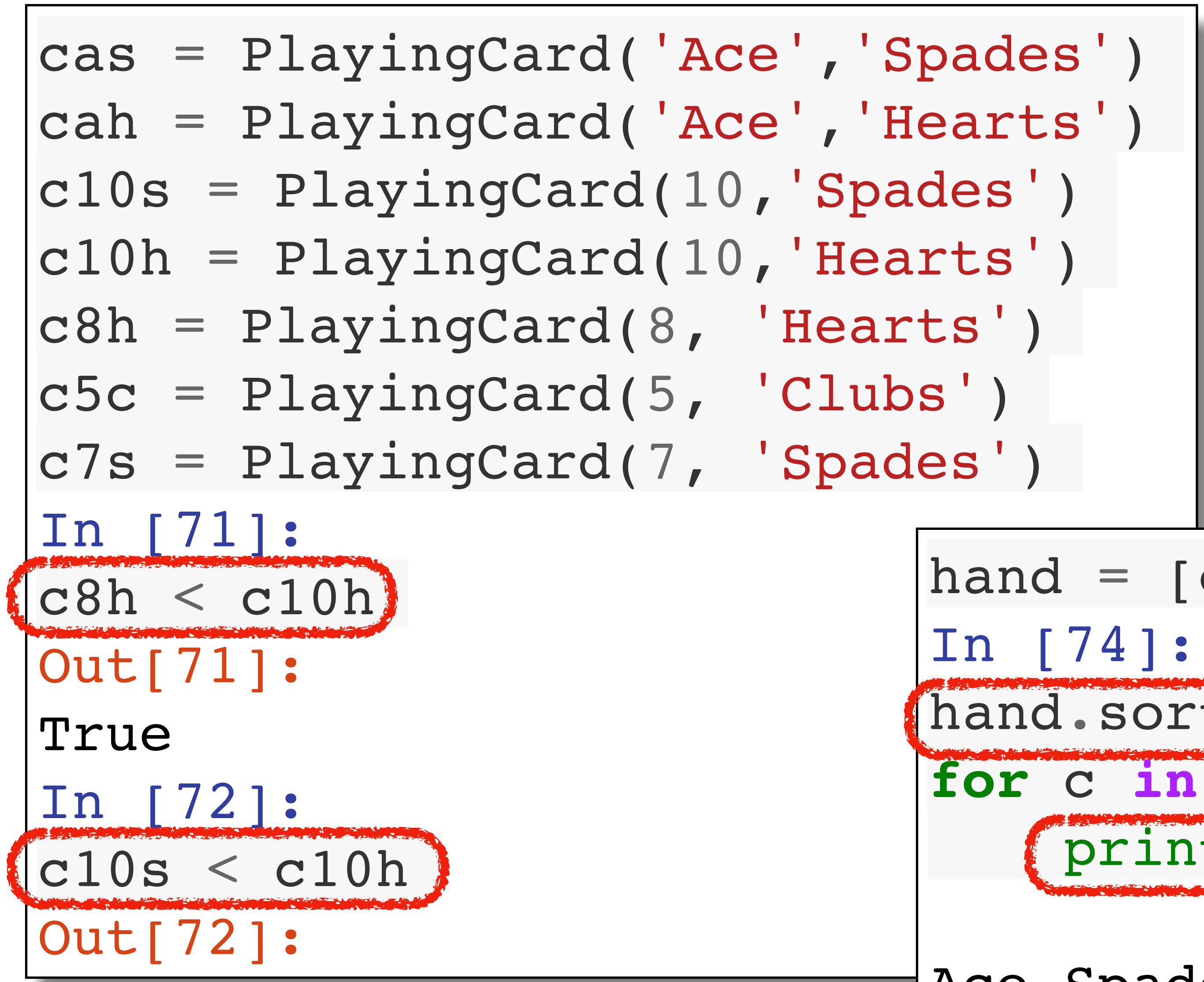
Greater Thank Method

- Add a `__gt__` method that will work as follows:
  - `t1 > t2`
  - `False`
- It sums the chains using `sum(self.chain)`

Example with < and str

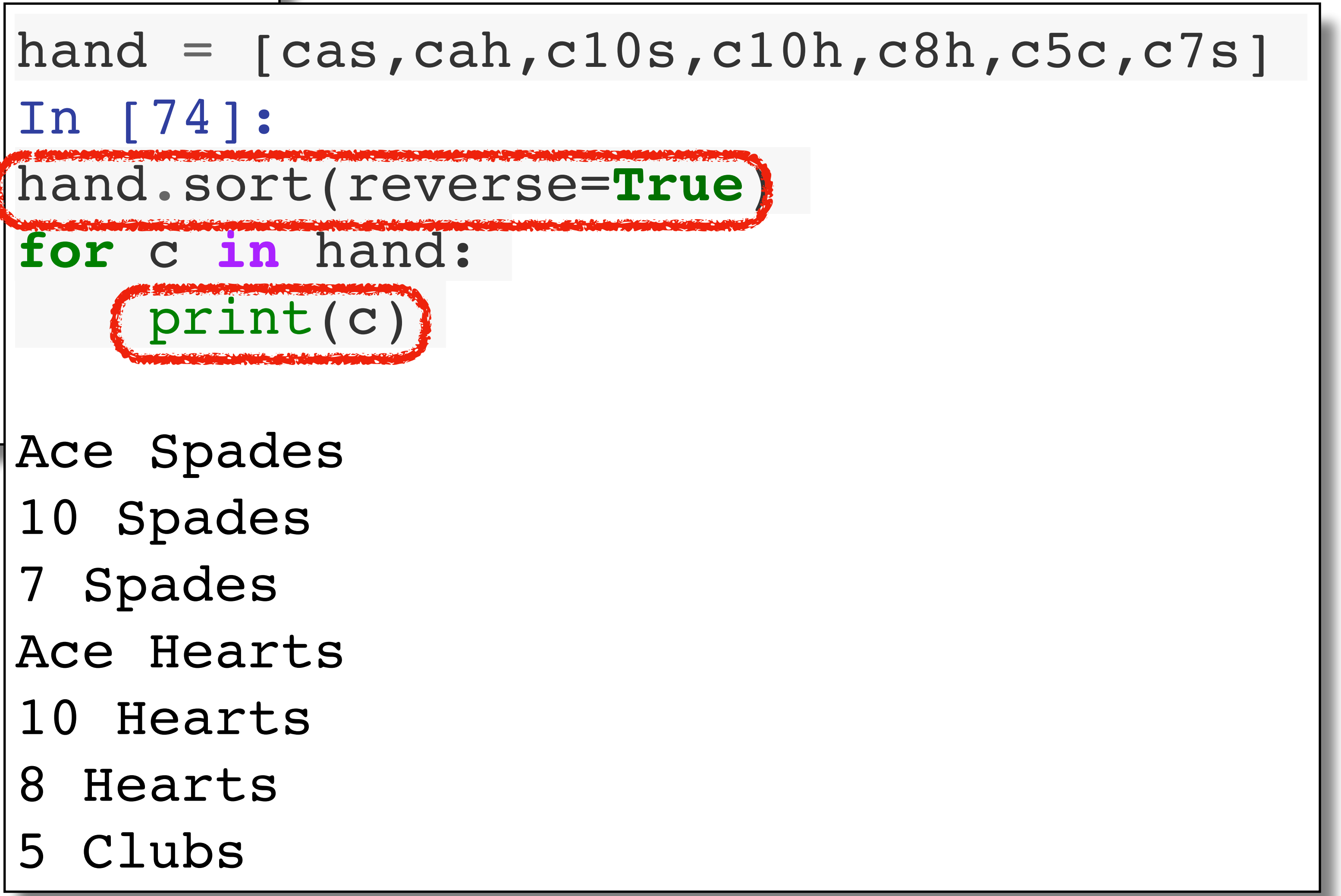
```python
suits = ['Spades','Hearts','Diamonds','Clubs']
rank = ['Ace','King','Queen','Jack',10,9,8,7,6,5,4,3,2]

class PlayingCard():
    rank_num = {'Ace':14,'King':13,'Queen':12,'Jack':11,10:10,
                9:9,8:8,7:7,6:6,5:5,4:4,3:3,2:2}

    def __init__(self,r,s):
        self.suit = s
        self.rank = r

    def __lt__(self,other):  # Only useful for sorting
        return (self.suit, self.rank_num[self.rank]) \
                < (other.suit, self.rank_num[other.rank])

    def __str__(self):
        return str(self.rank) + " " + self.suit

    def show(self):
        print(self.rank, self.suit)
```

Example with < and str



```
cas = PlayingCard('Ace','Spades')
cah = PlayingCard('Ace','Hearts')
c10s = PlayingCard(10,'Spades')
c10h = PlayingCard(10,'Hearts')
c8h = PlayingCard(8, 'Hearts')
c5c = PlayingCard(5, 'Clubs')
c7s = PlayingCard(7, 'Spades')
In [71]:
c8h < c10h
Out[71]:
True
In [72]:
c10s < c10h
Out[72]:
```

- __lt__ method enables
  - < operator
  - sorting
- __str__ enables print



```
hand = [cas,cah,c10s,c10h,c8h,c5c,c7s]
In [74]:
hand.sort(reverse=True)
for c in hand:
    print(c)

Ace Spades
10 Spades
7 Spades
Ace Hearts
10 Hearts
8 Hearts
5 Clubs
```

Sorting objects

- `hand.sort(reverse=True)` works because < operation is defined for the objects in the `hand` list.
- The other way to do this is to use a **lambda function**

Fun Fact

Alan Turing

Alonzo Church

- Lambda Functions are Anonymous functions
  - i.e. no name (handle)
  - function definition not bound to an identifier
- Name comes from Lambda Calculus - mathematics developed by Alonzo Church
- Fun Fact: Alonzo Church famous for Church - Turing thesis
  - hypothesis about the nature of computable functions
    - what can be computed
  - Turing - Turing Machines
  - Church - Lambda Calculus
  - Developed independently

Lambda Functions

- Simple, light weight functions
- No name
- In Python
  - very simple, restricted to one expression
- Relevance for OOP?
  - useful for accessing data within an object, e,g, for sorting or filtering

Remember myMapper from Lecture 2

- myMapper maps (applies) a function to all the elements in a list in turn.
- There is a built-in map function called (you guessed it) **map**
  - returns a map object
  - convert the map into a list
- We use the name of the function to pass it to the mapper

```
def square(e):
    return e*e

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

myMapper(t,square)
[1089, 1936, 3025, 4356]

map(square,t)
<map at 0x1108ad780>

list(map(square,t))
[1089, 1936, 3025, 4356]
```

Lambda functions

- Don’t bother defining the function and naming it
- Declare it inline in the map call

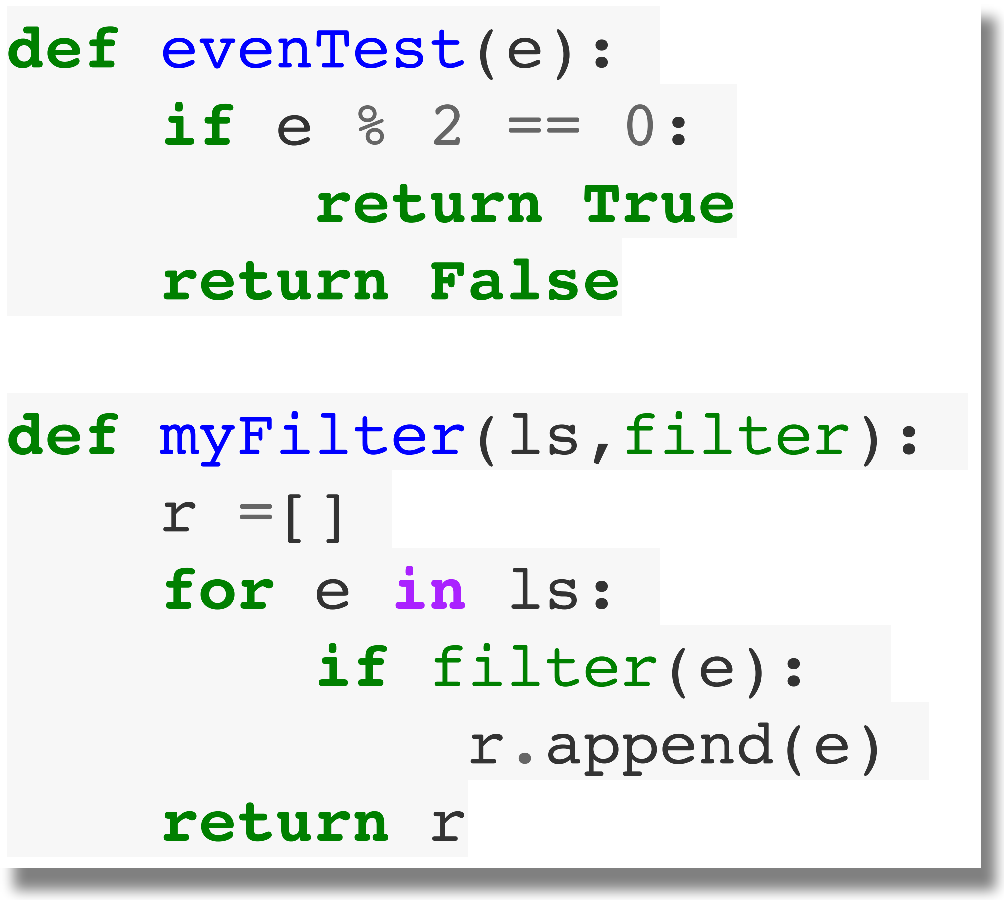
```
myMapper(t,lambda x:x*x)
Out[6]:
[1089, 1936, 3025, 4356]
In [15]:
map(lambda x:x*x,t)
Out[15]:
<map at 0x1109144e0>
In [13]:
list(map(lambda x:x*x,t))
Out[13]:
[1089, 1936, 3025, 4356]
```

The lambda function cannot be too complex - a single expression

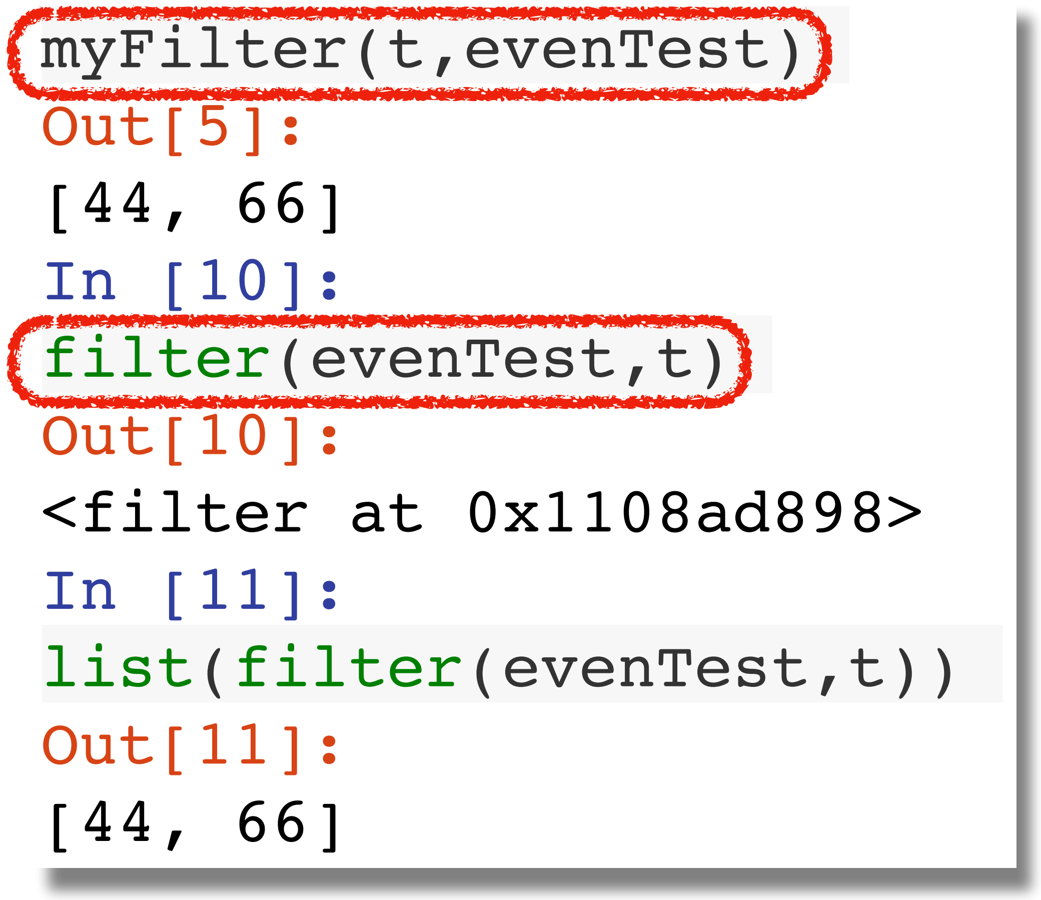
```
C = [39.2, 36.5, 37.3, 38, 37.9]
F = list(map(lambda x: (float(9)/5)*x + 32, C))
F
Out[21]:
[102.56, 97.7, 99.14, 100.4, 100.22]
```

Filter

- Remember the myFilter function
- There is also a built-in filter function.



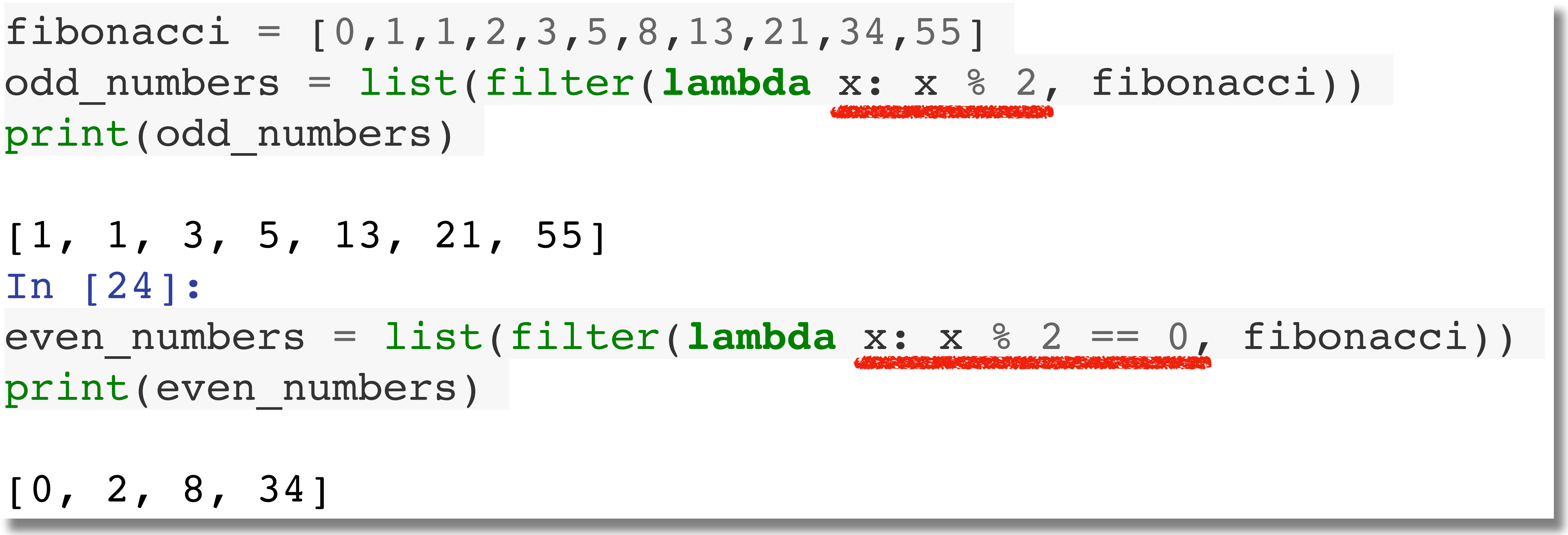
```
def evenTest(e):
    if e % 2 == 0:
        return True
    return False

def myFilter(ls,filter):
    r =[]
    for e in ls:
        if filter(e):
            r.append(e)
    return r
```



```
myFilter(t,evenTest)
Out[5]:
[44, 66]
In [10]:
filter(evenTest,t)
Out[10]:
<filter at 0x1108ad898>
In [11]:
list(filter(evenTest,t))
Out[11]:
[44, 66]
```

More filters

- Pull odd/even numbers from a list



```
fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
odd_numbers = list(filter(lambda x: x % 2, fibonacci))
print(odd_numbers)

[1, 1, 3, 5, 13, 21, 55]
In [24]:
even_numbers = list(filter(lambda x: x % 2 == 0, fibonacci))
print(even_numbers)

[0, 2, 8, 34]
```

Is lambda Pythonic?

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

```python
class Person():
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def __str__(self):
        return self.name +" "+ "Age:" \
            + str(self.age) + " Weight:" \
            + str(self.weight)
```

Back to Sorting

- Use a lambda function to access an attribute for sorting.
- Use the `__str__` method for pretty printing.

```python
b = Person("Betty", 45, 68)
j = Person("Jane", 34, 70)
m = Person("Mark", 23, 80)
s = Person("Sam", 25, 85)
In [33]:
gang = [m,j,s,b]
In [34]:
gang.sort(key=lambda x:x.weight,
          reverse = True)
for i in gang:
    print(i)
```

```
Sam Age:25 Weight:85
Mark Age:23 Weight:80
Jane Age:34 Weight:70
Betty Age:45 Weight:68
```

Overloading & Special Methods

- **Method Overloading**
  - A class can have more than one **method** having the same name
  - if their argument lists are different.
  - Not possible in Python because it is ***dynamically typed***
- **Special Methods**
  - Dunders - double underscores
  - `__init__`
  - `__str__`
  - `__lt__`
  - `__len__`
  - `__add__`
- **Lambda functions**