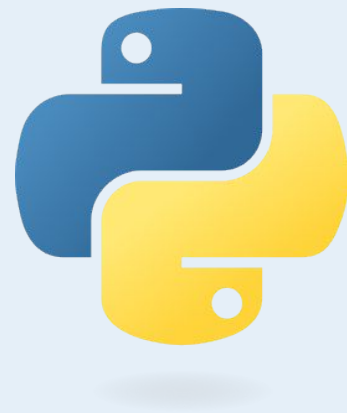




Object Oriented Programming

Python 101 - Week 5



Mehmet Arif Demirtaş
marifdemirtas.github.io

Recap: Data Types

Data Type	Literals
int	1, -4, 0, 10000
float	1.5, -3.2, 9.25, 15.0
bool	True, False
NoneType	None
str	'a', "abcd", 'hello world!'
tuple	(1,), ('a', 3, 8), ((1, 2), 3)
range	range(start, stop, step)
list	[1], ['a', 3, 8], [[1, 2], 3]
function	...

Numeric types
Supports addition, multiplication

Supports logical comparisons (<, >)

Placeholder for nothing

Scalars

Text

Items of different types

Numerical ranges

Items of different types, can be mutated

Functions are data types too!

Collections

Recap: Control Flow Structures

Conditional (if-else)	while (loop)	for (loop)
<pre>test1 = (5<3) and (10>2) test2 = (1!=1) or (not False) if test1: print("Test is true") elif test2: print("Test2 is true") else: print("Both are false")</pre>	<pre>i = 0 N = 6 while i < N: print(i * 2 + 1) i += 2</pre>	<pre>for num in range(0, 6, 2): print(num * 2 + 1)</pre>
Test2 is true	1 5 9	

Recap: Functions

Function Definition	Function Call	Result
<pre>def func1(): print("This is func 1")</pre>	<pre>x = 5 func1() x += 1</pre>	This is func 1
<pre>def func2(x): print(x ** 3)</pre>	<pre>a = 4 func2(7)</pre>	64
<pre>def func3(x, y): return x + 2 * y</pre>	<pre>a, b = 3, 5 c = func(a, b) print(c)</pre>	13
<pre>from math import sqrt as sq def func4(t1, t2, sqrt=True): res = t1**2 + t2**2 if sqrt: res = sq(res) return res</pre>	<pre>a, b = 3, 4 c1 = func4(a, b) print(f"Default: {c1}") c2 = func4(a, b, sqrt=False) print(f"No sqrt: {c2}")</pre>	Default: 5.0 No sqrt: 13

Recap: Structure of a data type

Object int:

```
name = a_number # variable name  
data = 0x010 # address
```

functions:

```
__add__: # add (+)  
... # statements that will sum up two numbers  
__mul__: # multiply (*)  
... # statements that will multiply two numbers
```

Object str:

```
name = a_string # variable name  
data_start = 0x010 # start address  
data_end = 0x030 # end address
```

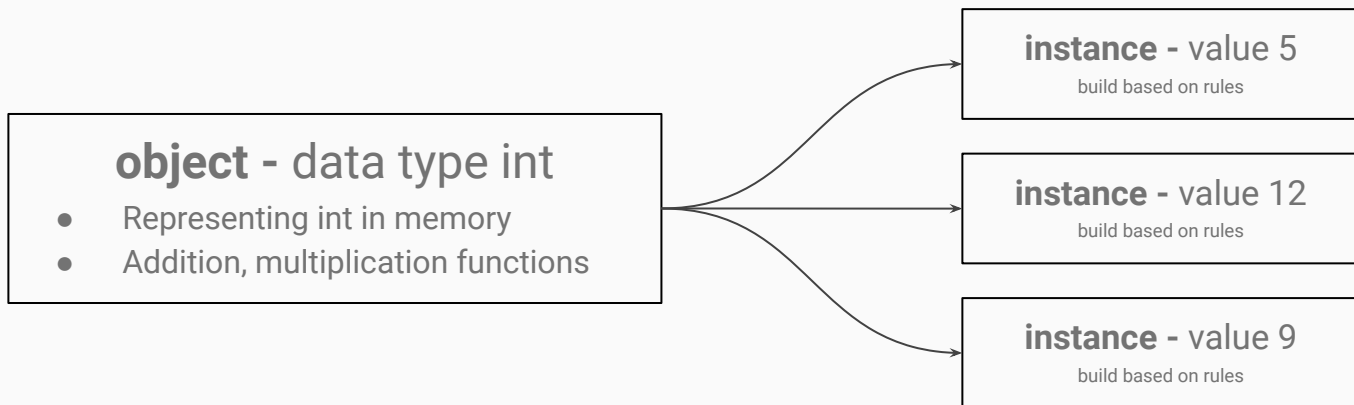
functions:

```
__add__: # add (+)  
... # statements that will concatenate strings  
__mul__: # multiply (*)  
... # statements that will concatenate a string with itself N times
```

Structure of an object

- Internal data representation
- Functions for interacting with the object

An **object** is the recipe/blueprint for the **instance** of that type.



Why use objects?

- Collect a concept and everything you can do with that concept under a single structure
- Abstract away inner representation and interact with well-defined interface functions (methods on an object)

Example: Fraction

```
import math
class Fraction(object):
    ...
    def __sub__(self, other):
        other.num *= -1
        return self + other
```

Define subtraction on Fractions by multiplying numerator by -1.

Methods

- `__init__`
- `__str__`
- `simplify`
- `__add__`
- `__sub__`

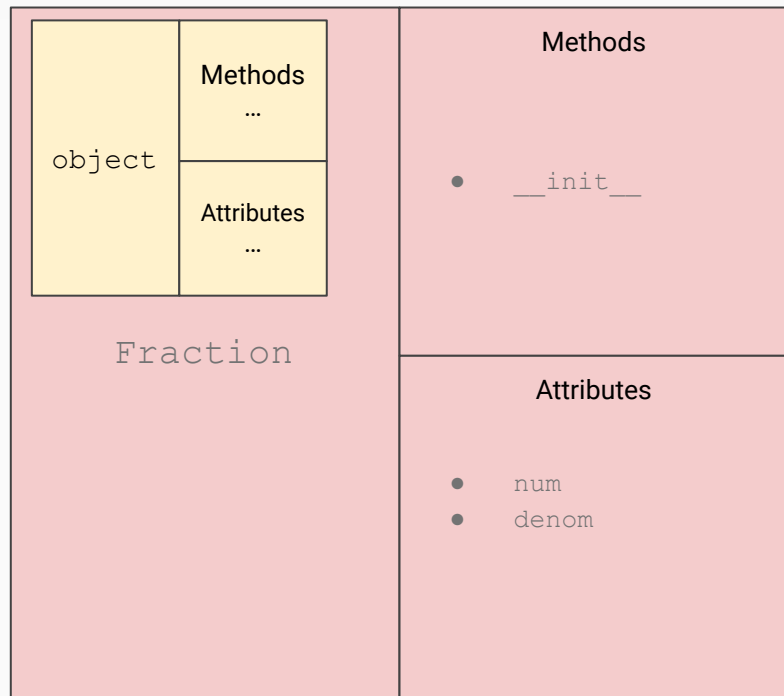
Attributes

- `num`
- `denom`

Example: Fraction

```
class Fraction(object):  
    def __init__(self,  
                  num,  
                  denom):  
        self.num = num  
        self.denom = denom
```

class: Start an object recipe definition
(object): **Inherit** from a parent
__init__: Define a **constructor** function
self: This is a class function, so it takes the object instance as first parameter.
self.num: The instance now has an **attribute** called num.



Example: Fraction - special methods

```
class Fraction(object):  
    ...  
    def __str__(self):  
        return f"{self.num}/{self.denom}"
```

Define a special methods starting with double underscores that will be called by Python:

String representation of the instance

Methods

- `__init__`
- `__str__`

Attributes

- `num`
- `denom`

Example: Fraction

```
import math
class Fraction(object):
    ...
    def simplify(self):
        gcd = math.gcd(self.num, self.denom)
        self.num /= gcd
        self.denom /= gcd
```

Simplify a fraction by importing **gcd** function and dividing numerator and denominator by gcd of two values.

Methods

- `__init__`
- `__str__`
- `simplify`

Attributes

- `num`
- `denom`

Example: Fraction

```
import math
class Fraction(object):
    ...
    def __add__(self, other):
        result = Fraction(
            num=(self.num * other.denom +
                 self.denom * other.num),
            denom=(self.denom * other.denom)
        )
        result.simplify()
        return result
```

Define addition on Fractions.

Methods

- `__init__`
- `__str__`
- `simplify`
- `__add__`

Attributes

- `num`
- `denom`

Example: Fraction

```
class Fraction(object):
    ...
    def __add__(self, other):
        if isinstance(other, int):
            other = Fraction(other, 1)
        result = Fraction(
            num=(self.num * other.denom +
                 self.denom * other.num),
            denom=(self.denom * other.denom)
        )
        result.simplify()
        return result
```

Methods

- `__init__`
- `__str__`
- `simplify`
- `__add__`

Attributes

- `num`
- `denom`

Example: Fraction

```
import math
class Fraction(object):
    ...
    def __mul__(self, other):
        TODO
```

Define multiplication on fractions.

Methods

- `__init__`
- `__str__`
- `simplify`
- `__add__`
- `__sub__`

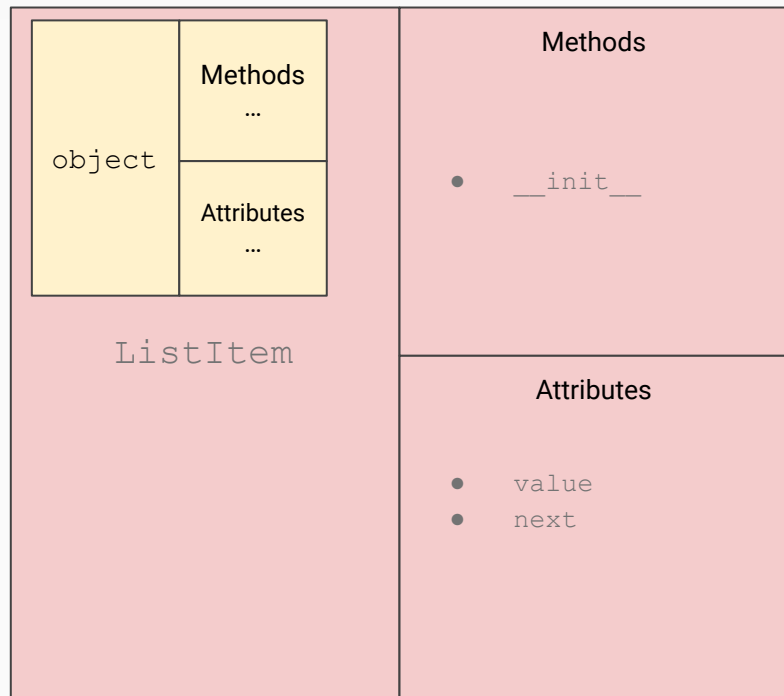
Attributes

- `num`
- `denom`

Example: Lists from scratch

```
class ListItem(object):  
    def __init__(self, value, next=None):  
        self.value = value  
        self.next = next
```

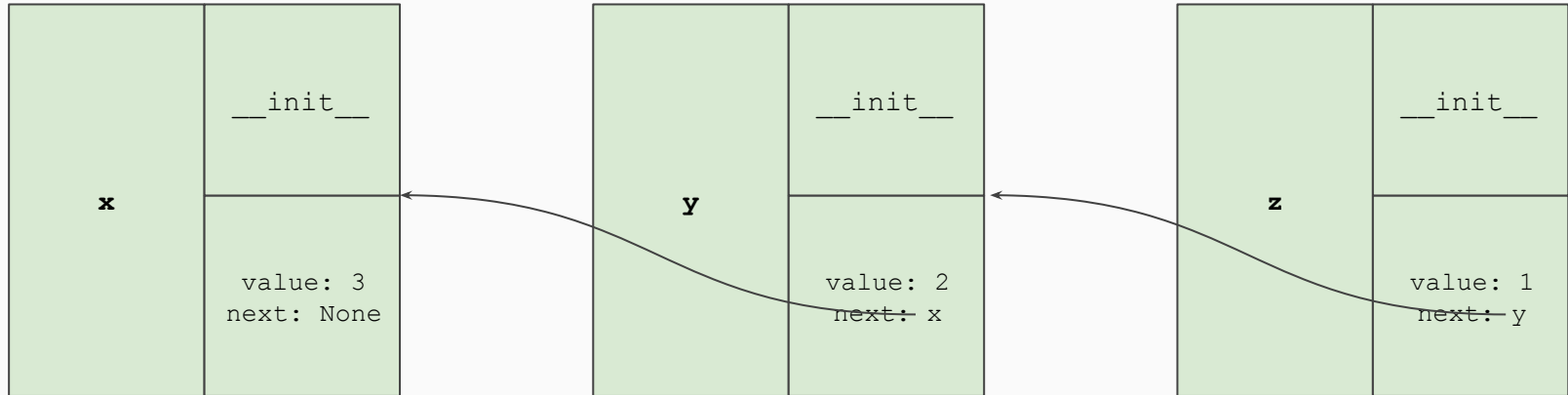
class: Start an object recipe definition
(object): **Inherit** from a parent
__init__: Define a **constructor** function
self: This is a class function, so it takes the object instance as first parameter.
self.value: The instance now has an **attribute** called value.



Example: Lists from scratch

```
x = ListItem(3)
y = ListItem(2, x)
z = ListItem(1, y)
```

calls `__init__` function



Example: Lists from scratch

```
class List(object):  
    def __init__(self, head):  
        self.head = head  
  
    def __str__(self):  
        temp = self.head  
        while temp is not None:  
            print(temp.value)  
            temp = temp.next
```

List object:

- Assign a reference to the list head item
- Print the value attribute of the head item
- Move temp reference to the next attribute

```
my_list = List(z)  
my_list.print_list()
```

Example: Lists from scratch (modified)

```
class List(object):  
    def __init__(self, value):  
        self.head = ListItem(value)  
        self.tail = self.head
```

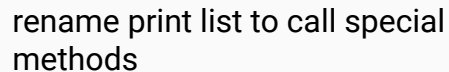
← keep a reference to the end

```
    def __str__(self):  
        temp = self.head  
        while temp is not None:  
            print(temp.value)  
            temp = temp.next
```

Example: Lists from scratch (special methods)

```
class List(object):  
    def __init__(self, value):  
        self.head = ListItem(value)  
        self.tail = self.head
```

```
    def __str__(self):  
        temp = self.head  
        while temp is not None:  
            print(temp.value)  
            temp = temp.next
```



rename print list to call special
methods

Example: Lists from scratch

```
class List(object):  
    ...  
  
    def append(self, value):  
        new_item = ListItem(value)  
        self.tail.next = new_item  
        self.tail = new_item
```

```
my_list = List(1)  
my_list.append(2)  
my_list.print_list()
```

Example: Lists from scratch

```
class List(object):  
    ...  
  
    def len(self):  
        len_list = 0  
        temp = self.head  
        while temp is not None:  
            len_list += 1  
            temp = temp.next  
        return len_list
```

```
print(my_list.len())
```

Example: Lists from scratch

```
class List(object):  
    ...  
    def getitem(self, index):  
        if index >= self.len() or index <  
0:  
            print("ERROR")  
        else:  
            temp = self.head  
            i = 0  
            while i < index:  
                temp = temp.next  
                i += 1  
            return temp.value
```

Example: Lists from scratch - special methods

```
class List(object):  
    ...  
    def __getitem__(self, index):  
        if index >= self.len() or index <  
0:  
            print("ERROR")  
        else:  
            temp = self.head  
            i = 0  
            while i < index:  
                temp = temp.next  
                i += 1  
            return temp.value
```

Example: Mutable list

```
class MutableList(List):  
    def __init__(self, value):  
        super().__init__(value)
```

Child of regular List class

Call parent constructor

List	Methods <ul style="list-style-type: none">• <code>print_list</code>• <code>append</code>• <code>__str__</code>• <code>len</code>• <code>__getitem__</code>
MutableList	<ul style="list-style-type: none">• <code>__init__</code>• <code>__setitem__</code>
	Attributes <ul style="list-style-type: none">• <code>value</code>• <code>next</code>

Example: Mutable list

```
class MutableList(List):  
    ...
```

← *Child of regular List class*

```
def __setitem__(self, index, value):  
    if index >= self.len() or index < 0:  
        print("ERROR")  
    else:  
        temp = self.head  
        i = 0  
        while i < index:  
            temp = temp.next  
            i += 1  
        temp.value = value
```

← *Call parent constructor*

Example: Mutable list

```
class MutableList(List):  
    ...  
  
    def __add__(self, other):  
        TODO  
  
    def __mul__(self, times):  
        TODO using __add__
```

Remember that:

- `__add__` stands for (+) and it concatenates two lists:
 $[1, 2] + [3, 4] = [1, 2, 3, 4]$
- `__mul__` stands for (*) and it concatenates a list with itself N times:
 $[1, 2] * 3 = [1, 2, 1, 2, 1, 2]$