

CptS 355- Programming Language Design

Python

Instructor: Sakire Arslan Ay
Fall 2023



World Class. Face to Face.

Lecture material

- Please watch the Python part-1 and part2 videos on Canvas.
- No lecture notes on Python basics
- Lecture notes on:
 - Python lists
 - Python dictionaries
 - Higher order functions
 - **Classes, iterators, streams**

The Class Statement

```
class <name> <base class>  
    <suite>
```

- A class statement creates a new class and binds that class to **<name>** in the current environment.
- Statements in the **<suite>** create attributes of the class.
- As soon as an instance is created, it is passed to `__init__`, which is an attribute of the class.

```
class Account(object):  
    def __init__(self, account_holder):  
        self.balance = 0  
        self.holder = account_holder
```

The Class Statement

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

- Classes are "called" to construct instances.
- The constructor `__init__` is called on newly created instances.
- The new object is bound to `__init__`'s first parameter, `self`.

```
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

Object Identity

- Every object that is an instance of a user-defined class has a unique identity

```
>>> a = Account( 'Sam' )  
>>> b = Account( 'Sam' )
```

- Identity testing is performed by "is" and "is not" operators:

```
>>> a is b  
False  
>>> a is not b  
True
```

- Binding an object to a new name using assignment does not create a new object:

```
>>> c = a  
>>> c is a  
True
```

Class Methods

- Methods are defined in the suite of a class statement

```
class Account(object):  
    def __init__(self, account_holder):  
        self.balance = 0  
        self.holder = account_holder  
  
    def deposit(self, amount):  
        self.balance = self.balance + amount  
        return self.balance  
  
    def withdraw(self, amount):  
        if amount > self.balance:  
            return 'Insufficient funds'  
        self.balance = self.balance - amount  
        return self.balance
```

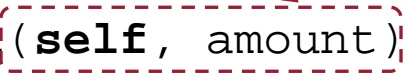
- These `def` statements create function objects as always, but their names are bound as attributes of the class.

Invoking Methods

- All invoked methods have access to the object via the `self` parameter, and so they can all access and manipulate the object's state.

```
class Account(object):  
    ...  
    def deposit(self, amount):  
        self.balance = self.balance + amount  
        return self.balance
```

two arguments



- Dot notation automatically supplies the first argument to a method.

```
>>> sam_account = Account('Sam')  
>>> sam_account.deposit(100)  
100
```

Invoked with one argument



Methods and Functions

- Python distinguishes between:
 - *function objects*, which we have been creating since the beginning of the course, and
 - *bound method objects*, which couple together a function and the object on which that method will be invoked

```
>>> type(Account.deposit)
```

```
<class 'function'>
```

```
>>> type(sam_account.deposit)
```

```
<class 'method'>
```

```
>>> Account.deposit(sam_account, 1001)
```

```
1011
```

```
>>> sam_account.deposit(1000)
```

```
2011
```


Class @property

```
class Account(object):  
    def __init__(self, account_holder):  
        self.balance = 0  
        self.holder = account_holder  
  
    def deposit(self, amount):  
        self.balance = self.balance + amount  
        return self.balance  
  
    def withdraw(self, amount):  
        if amount > self.balance:  
            return 'Insufficient funds'  
        self.balance = self.balance - amount  
        return self.balance
```

```
>>> sam_account = Account('Sam')  
>>> sam_account.balance = -100
```

- We don't want the balance property to go below 0.
- A solution to this will be to hide the attribute balance (make it private) and define new getter and setter interfaces to manipulate it

Python Class @property

```
class Account(object):
    def __init__(self, account_holder):
        self.set_balance(0):
        self.holder = account_holder

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance: return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance

    def get_balance(self):
        return self._balance

    def set_balance(self, amount):
        if amount < 0:
            raise ValueError("Balance can't be negative")
        else:
            self._balance = amount

    balance = property(get_balance, set_balance)

>>> sam_account = Account('Sam')
>>> sam_account.balance = -100
```

Sequential Data

- Some of the most interesting real-world problems in computer science center around sequential data.
 - DNA sequences
 - Web and cell-phone traffic streams
 - The social data stream
 - Series of measurements from instruments on a robot
 - Stock prices, weather patterns

Working with Sequences

- Memory
 - Each item must be explicitly represented
 - Even if all can be generated by a common formula or function
- Up-front computation
 - Have to compute all items up-front
 - Even if using them one by one
 - Can't be infinite
- Why care about “infinite” sequences?
 - They're everywhere!
 - Internet and cell phone traffic
 - Instrument measurement feeds, real-time data
 - Mathematical sequences

Iterators: An abstraction for sequential data

- Iterators
 - Store how to compute items instead of items themselves
 - Give out one item at a time
 - Save the next until asked (lazy evaluation)
- Compared with sequences
 - Length not explicitly defined
 - No up-front computation of all items
 - Only one item stored at a time
 - Can be infinite
 - Element selection not supported
 - Sequences -- random access
 - Iterators -- sequential access

Lazy Evaluation

- Lazy evaluation is a strategy implemented in some programming languages which delays the evaluation of an expression until its value is needed.
 - If you only need to iterate through a sequence of objects that can be generated and acted upon one at a time, then you do not need to explicitly construct a big container to process the elements one-by-one.

Python Iterators

- The Python iterator interface includes two attributes:
 - The `__next__` attribute compute the next element in an underlying series.
 - Calls to `__next__` advance the position of the iterator.
 - Python signals that the end of an underlying series has been reached by raising a `StopIteration` exception during a call to `__next__`.
 - The `__iter__` message simply returns the iterator.
 - It is useful for providing a common interface to iterators and sequences

Iterator Example - 1

```
class Letters(object):  
    def __init__(self):  
        self.current = 'a'  
  
    def __next__(self):  
        if self.current > 'd':  
            raise StopIteration  
        result = self.current  
        self.current = chr(ord(result)+1)  
        return result  
  
    def __iter__(self):  
        return self
```

```
>>> letters = Letters()  
>>> letters.__next__()  
'a'  
>>> letters.__next__()  
'b'  
>>> letters.__next__()  
'c'  
>>> letters.__next__()  
'd'  
>>> letters.__next__()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in  
<module>  
    File "<stdin>", line 12, in next  
StopIteration
```

- A Letters instance can only be iterated through once.
- Once its `__next__()` method raises a `StopIteration` exception, it continues to do so from then on.
- There is no way to reset it; one must create a new instance.

Iterator Example - 2

```
class Evens(object):  
    def __init__(self):  
        self.current = 2  
  
    def __next__(self):  
        result = self.current  
        self.current += 2  
        return result  
  
    def __iter__(self):  
        return self
```

```
evenNums= Evens()  
n = 10  
while n > 0:  
    item = evenNums.__next__()  
    print(item)  
    n -= 1
```

This prints even numbers 2 through 20.

- Iterators also allow us to represent infinite series by implementing a `__next__` method that never raises a `StopIteration` exception

Native Python Iterators

- Python natively supports iterators
- The Iterator interface in python:
 - `__iter__` : should return an iterator object
 - `__next__` : should return a value OR raise `StopIteration` when end of sequence is reached on all subsequent calls
- In Python, most of built-in containers like: list, tuple, string, dictionary, etc. are iterables.
 - The `iter()` function (which in turn calls the `__iter__()` method) returns an iterator from them.

for statements and Iterators

- The iterator returned by invoking the `__iter__` method of `counts` is bound to a name `i` so that it can be queried for each element in turn

```
>>> counts = [1, 2, 3]
>>> for item in counts:
    print(item)
```

=

```
>>> counts = [1, 2, 3]
>>> i = counts.__iter__()
>>> try:
    while True:
        item = i.__next__()
        print(item)
    except StopIteration:
        pass
```

Python Generators

- The `Letters` iterator can be implemented much more compactly using a generator function.

Iterator version

```
class Letters(object):  
    def __init__(self):  
        self.current = 'a'  
  
    def __next__(self):  
        if self.current > 'd':  
            raise StopIteration  
        result = self.current  
        self.current = chr(ord(result)+1)  
        return result  
  
    def __iter__(self):  
        return self
```

Generator version

```
def letters(start, finish):  
    current = start  
    while current <= finish:  
        yield current  
        current = chr(ord(current)+1)
```

Yield: a built-in flow-control statement

```
def letters(start, finish):  
    current = start  
    while current <= finish:  
        yield current  
        current = chr(ord(current)+1)
```

Generator function.

When called, creates a Generator object

```
>>> g = letters('a', 'd')  
>>> g  
<generator instance at...>
```

The generator function:

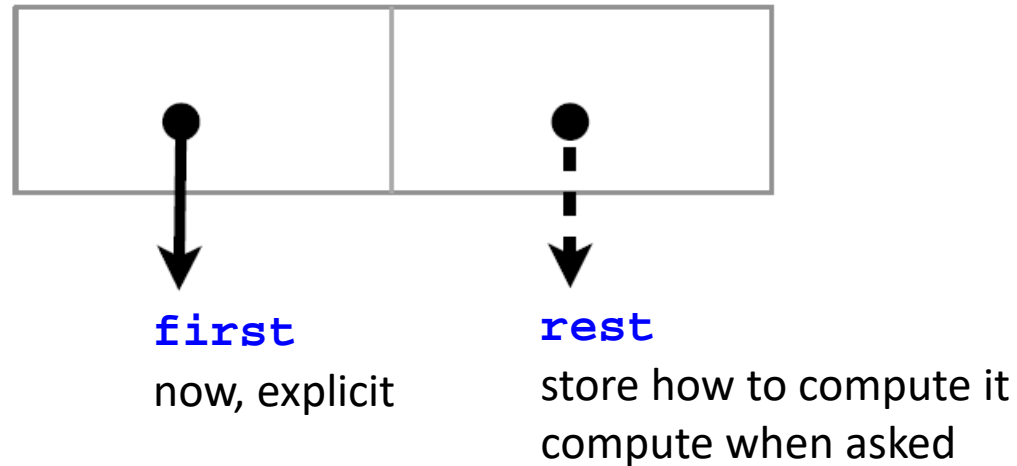
- Does nothing at first
- When `__next__()` is called, starts
- Goes through executing body of function
- Pauses at “yield” -- returns value
- All local state is preserved
- When `__next__()` is called, resumes.

Automatically creates:

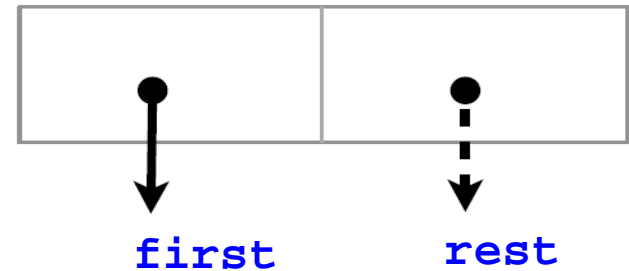
```
g.__iter__()  
g.__next__()
```

Streams: A Lazy Structure

- Nested delayed evaluation



Streams

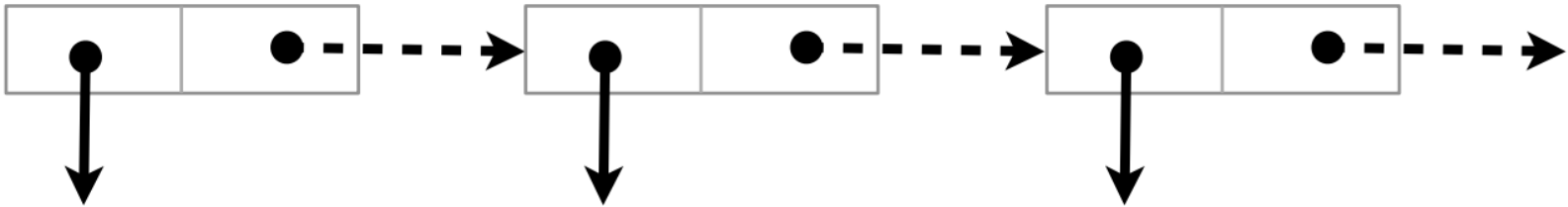


```
class Stream(object):
    def __init__(self, first, compute_rest, empty= False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

    @property
    def rest(self):
        assert not self.empty, 'Empty streams have no rest.'
        if not self._computed:
            self._rest = self._compute_rest()
            self._computed = True
        return self._rest

empty_stream = Stream(None, None, True)
```

Sequential data as nested streams



- Nest streams inside each other
- Only compute one element of a sequence at a time
- Example: The positive integers (all of them)

```
def make_integer_stream(first=1):  
    def compute_rest():  
        return make_integer_stream(first+1)  
    return Stream(first, compute_rest)
```

```
>>> N = make_integer_stream(1)  
>>> N.first  
1  
>>> N = N.rest  
>>> N.first  
2  
>>> N.rest.first  
3
```


Streams in Action

- Initially, `L=make_integer_stream(1)` consists of one item with
`L.first = 1, L._computed = False`
- When we fetch `L.rest`, it becomes
`L.first = 1, L._computed = True;`
`L._rest = make_integer_stream(2),`
`# where`
`L._rest.first = 2, L._rest._computed = False`
- And so forth.

```
def make_integer_stream(first=1):  
    def compute_rest():  
        return make_integer_stream(first+1)  
    return Stream(first, compute_rest)
```

Stream Examples

- Double the stream

```
def double_stream(s):  
    if s.empty:  
        return s  
    def compute_rest():  
        return double_stream(s.rest)  
    return Stream(2*(s.first), compute_rest)
```

Stream Examples

- Combine streams

```
def combine_streams(fn, s0, s1):  
    def compute_rest():  
        return combine_streams(fn, s0.rest, s1.rest)  
    if s0.empty or s1.empty:  
        return empty_stream  
    else:  
        return Stream(fn(s0.first, s1.first), compute_rest)
```