# THE PYTHON PROGRAMMING LANGUAGE

Part 2: Intermediate Topics

*April 28, 2020*

Hemanth P. Sethuram

**Email:** hemanth.p.sethuram@intel.com

# Contents

# Contents ...2

- Other Ways to Iterate
  - map
  - filter
  - reduce
- Generators
- Comprehensions
- Complex example showing various concepts
- Context Managers for Automatic Resource Management
- Decorators
  - Definition
  - Usage
  - Applications

# Python Data Model

# Operator Overloading

- **Python** provides a rich data model to allow user defined operator overloading facilities.

- Look at `https://docs.python.org/3/reference/datamodel.html`.

- By providing implementation to these special methods of the `object`, one can provide a standard set of features similar to built-in objects.

# Making objects comparable

You can make your class instances comparable by defining the following special methods in your class definitions.

- `__lt__(self, other)`
- `__le__(self, other)`
- `__eq__(self, other)`
- `__ne__(self, other)`
- `__gt__(self, other)`
- `__ge__(self, other)`

# Customizing Attribute Access

- By default, all attributes of a class and its instance are **public**.
- One can modify the default access behavior by defining the following special methods.
- `__getattr__(self, name)` - called if regular attribute lookup fails. Useful for providing computed attributes.
- `__getattribute__(self, name)` - unconditionally called for all attribute accesses. If defined, `__getattr__()` is not called unless this function raises `AttributeError`.

# Customizing Attribute Access ...2

- `__setattr__(self, name, value)` - if defined, it is called when an attribute is assigned.
- `__delattr__(self, name)` - like `__setattr__` but called when an attribute is deleted.

# Emulating Container Types

Containers usually are sequences or mappings. The following methods can be defined to implement container objects.

- `__len__(self)` - Called by the built-in function `len()`.
- `__getitem__(self, key)` - `key` is an integer for sequences and any immutable Python object for mappings.
- `__setitem__(self, key, value)` - called when you do `self[key] = value`
- `__delitem__(self, key)`

# Emulating Container Types ...2

- `__iter__(self)` - optional, but desirable.
- `__reversed__(self)` - optional, but desirable. Should provide a reverse iterator.
- `__contains__(self, item)` - enables membership test with `in` operator.

# Emulating Numeric Types

When you define some of the following methods, your class instances will behave as a numeric type for the corresponding operation.

- `__add__(self, other)`
- `__sub__(self, other)`
- `__mul__(self, other)`
- `__truediv__(self, other)`
- `__floordiv__(self, other)`
- `__mod__(self, other)`
- `__divmod__(self, other)`
- `__pow__(self, other[, modulo])`
- `__lshift__(self, other)`
- `__rshift__(self, other)`

# Further Reading

See `https://docs.python.org/3/library/operator.html` for a complete list of operators that can be defined as functions.

# Function Arguments

# Arguments Destructuring

- Members in an iterable like a `list` or a `tuple` can be passed as positional arguments to a function by calling it as `func(*iterable)`.

- Contents of a dictionary can be passed as named arguments to a function by calling it as `func(**mydict)`.

```python
def func(a, b, c):
    return a + b + c  # a simple function

x = [3, 9, 8] # a list containing the arguments
print(func(*x)) # passes contents of list as positional arguments
d = { "a": 3, "b": 9, "c": 8} # dict having arguments
print(func(**d)) # passes contents as named arguments
```

# Variable number of arguments

- A function can be declared to collect variable number of positional arguments by using the `*args` notation in the function definition.

- Similarly, a function can be declared to collect variable number of named arguments by using the `**args` notation in the function definition.

```
def func(*args):
    print(args) # args is a tuple containing all positional arguments

def func(**kwargs):
    for key in kwargs: #kwargs is a dict containing named arguments
        print(key, ":", kwargs[key])
```

# Example with variable number of arguments

```
>>> def func(*args, **kwargs):
        print("Positional args: ", args) #prints list of positional args
        print("Named args:")
        for key in kwargs: #kwargs is a dict with named arguments
            print(key, ":", kwargs[key])
>>> func(3, 5, a=10, b=13, z=45) #call with both argument types
Positional args:  (3, 5)
Named args:
a : 10
b : 13
z : 45
>>>
```

# Iterators

# Defining an iterator

- An **iterator** is an object which can iterate over a collection in a defined order. The order depends on the implementation of the iterator.
- An iterator has to support 2 methods - `__iter__()` and `__next__()`.
- `__iter__()` returns the iterator object itself.
- `__next__()` returns the next object of the collection.

# Defining a Collection that is Iterable

- The collection which is iterable should also define an `__iter__(self)` method which returns a new iterator object that can iterate over all the objects in the container.

- If a `__reversed__(self)` method is defined in the collection class, it should return a new iterator object that supports reverse iteration. This method is called when `reversed(collection)` builtin is called.

# Calling an iterator

- An iterator to a collection can be obtained by calling the function `iter(collection)`. This internally calls `collection.__iter__()` to obtain an iterator.

- A reverse iterator to a collection can be obtained by calling the function `reversed(collection)`. This internally calls `collection.__reversed__()`, if the collection class has a definition for `__reversed__()` method. Else, a default reverse iterator object which uses `__getitem__()` and `__len__()` methods is returned.

- On the obtained iterator, you can successively get the next elements in 2 ways:
  - by calling `next(it)` successively. OR
  - by using the `for` loop as, `for item in it:`

# Example iterator code

```
class MyCollection():
    def __init__(self):
        self.x=[]
    def append(self, item):
        self.x.append(item)
    def __iter__(self):
        return MyIterator(self.x) # return a new iterator
```

```python
class MyIterator():
    def __init__(self, coll):
        self.coll = coll
        self.i = 0 # index
    def __iter__(self):
        return self
    def __next__(self):
        if self.i < len(self.coll):
            value = self.coll[self.i]
            self.i += 1
            return value
        else:
            raise StopIteration #for loop automatically terminates
```

# Example of iterators - continued

```
c = MyCollection() # create a collection
c.append(3)
c.append(15.4)
c.append("Hello")
c.append(10)
it1 = iter(c) # get an iterator to c
print(next(it1)) #calls it1.__next__() internally
print(next(it1))
for item in it1: # iterate using a for loop
    print(item)
print(next(it1)) # will raise StopIteration exception
it2 = iter(c) # get another iterator
print(next(it2)) # prints 3
for item in it2: # iterate on second iterator
    print(item)
```

# Further Reading

See `https://docs.python.org/3/library/itertools.html` for a list of rich and efficient iterator utilities.

# Other Ways to Iterate

# map

- `map(function, iterable, [iterable2, [iterable3,]],...)`
  Return an iterator that applies function to every item of iterable, yielding the results.

- If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel.

- With multiple iterables, the iterator stops when the shortest iterable is exhausted.

# map example

```
>> m = ["2.3", "4", "5.6", "1.4"]
>> n = list(map(float,m)) # [2.3, 4.0, 5.6, 1.4]
>> # should return [5.28999, 16.0, 31.35999, 1.95999]
>> squares = list(map(lambda x: x**2, n))
>> pow(2,10) #1024
>> pow(3,11) #177147
>> pow(4,12) #16777216
>> # should return [1024, 177147, 16777216]
>> list(map(pow,[2, 3, 4], [10, 11, 12]))
```

# filter

- `filter(function, iterable)` Construct an iterator from those elements of `iterable` for which function returns true.

- `iterable` may be either a sequence, a container which supports iteration, or an iterator.

- If function is `None`, the identity function is assumed, that is, all elements of `iterable` that are false are removed.

- Note that `filter(function, iterable)` is equivalent to the generator expression.
  - `(item for item in iterable if function(item))` if `function` is not `None` and
  - `(item for item in iterable if item)` if `function` is `None`.

# filter example

```
>> # [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>> list(range(-5,5))
>> evens = list(filter(lambda x: x%2 == 0, range(-5,5)))
>> # evens should be [-4, -2, 0, 2, 4]
>> data = [3, 5, 0, "Hello", "day", "", 23, [], None]
>> list(filter(None, data))
>> # should return [3, 5, 'Hello', 'day', 23]
>> # useful in skipping holes in external data
```

# reduce

- `functools.reduce(function, iterable[, initializer])`
- Apply function of two arguments cumulatively to the items of sequence, from left to right, so as to reduce the sequence to a single value.
- For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. The left argument, `x`, is the accumulated value and the right argument, `y`, is the update value from the sequence.
- If the optional initializer is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If initializer is not given and sequence contains only one item, the first item is returned.

# reduce example

```
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4], 20)
30
>>> reduce(lambda x, y: x*y, [2], 3)
6
>>> reduce(lambda x, y: x*y, [2])
2
```

# Further Reading

- See the `functools` module at `https://docs.python.org/3/library/functools.html`.
- This module is for **higher order functions**: functions that act on or return other functions.
- In general, any callable object (which defines `__call__()` method) can be treated as a function for this module.

# Generators

# Generator Functions

- Any function that contains a `yield` keyword is called a **generator function**.
- When the `yield` statement is executed, the control jumps from the function to the caller.
- After the caller communicates to the generator (through a `send` or a `next` command), the control jumps back to the point where the `yield` statement was executed.
- Generator functions can be used to get iterations instead of defining an iterator object.
- Generators can be shutdown using the `close()` method on them.

# Generator Function Example

```
>> def countdown(n):
>>     print("Counting down from %d" % n)
>>     while n > 0:
>>         yield n
>>         n -= 1
>>     return

>> g = countdown(10) # creates a generator object
>> dir(g) # displays methods of generator
>> next(g) # internally calls g.__next__(). First execution of g
>> next(g) # similar to iterator, move forward
>> g.send(25) # you can also send a value to move forward
>> for item in g: # you can also use for loop
>>     print(item)
>> next(g) # raises a StopIteration exception
```

# Generator Function as a co-routine Example

```
def line_splitter(delimiter=None):
    print("Ready to split")
    result = None
    while True:
        # yield result returns result
        # line gets what was sent using send(obj) from user
        line = (yield result)
        result = line.split(delimiter)

s = line_splitter(",") # creates a generator object and binds to s
next(s) # start the first step, internally calls s.__next__()
print(s.send("A,B,C")) # outputs ['A', 'B', 'C' ]
print(s.send("100,200,300")) # outputs ['100', '200', '300']
```

# Comprehensions

- Comprehension syntax allow expresssions to be used to create a collection of objects.
- Comprehensions can be used to create any container - a list, a dict, a set or a generator.

# Comprehensions Example

```python
# list comprehension. Notice the filter
cubes = [i**3 for i in range(15) if i%2 == 1]
type(cubes)
print(cubes)
#dict comprehension
d = {key:val for key,val in enumerate("ABCDE")}
type(d)
print(d)
#set comprehension: set of even numbers
s = {i for i in range(10) if i % 2 == 0}
type(s)
print(s)
```

```python
#Generator expression
g = (i**3 for i in range(13) if i % 2 == 1)
type(g)
print(g)
# Generator is evaluated only when needed
m = list(g)
```

# Example contrasting all approaches

```python
# generator
def uc_gen(text):
    for char in text:
        yield char.upper()

# generator expression
def uc_genexp(text):
    return (char.upper() for char in text)
```

# Example contrasting all approaches ..2

```python
# iterator protocol
class uc_iter():
    def __init__(self, text):
        self.text = text
        self.index = 0
    def __iter__(self):
        return self
    def __next__(self):
        try:
            result = self.text[self.index].upper()
        except IndexError:
            raise StopIteration
        self.index += 1
        return result
```

# Example contrasting all approaches ..3

```
 # getitem method
 class uc_getitem():
     def __init__(self, text):
         self.text = text
     def __getitem__(self, index):
         result = self.text[index].upper()
         return result

# To see all four methods in action, try this:
for iterator in uc_gen, uc_genexp, uc_iter, uc_getitem:
    for ch in iterator('abcde'):
        print(ch, end=" ")
    print()
# The output should be
A B C D E
A B C D E
A B C D E
A B C D E
```

# Example: Processing Pipelines with Generators ..1

```python
import os
import fnmatch

def find_files(topdir, pattern):
    for path, dirname, filelist in os.walk(topdir):
        for name in filelist:
            if fnmatch.fnmatch(name, pattern):
                yield os.path.join(path,name)

import gzip, bz2
def opener(filenames):
    for name in filenames:
        if name.endswith(".gz"):  f = gzip.open(name)
        elif name.endswith(".bz2"): f = bz2.BZ2File(name)
        else: f = open(name)
        yield f
```

# Example: Processing Pipelines with Generators ..2

```
def cat(filelist):
    for f in filelist:
        for line in f:
            yield line

def grep(pattern, lines):
    for line in lines:
        if pattern in line:
            yield line
```

# Example: Processing Pipelines with Generators ..3

```python
# create generators and chain them sequentially
wwwlogs = find("www","access-log*")
files   = opener(wwwlogs)
lines   = cat(files)
pylines = grep("python", lines)

# Until now, we only have a description of how the
# objects have to be created, but no actual creation.
# iterate over the last generator
for line in pylines:
    # at this point all the chained generators are
    # executed and the objects are created
    sys.stdout.write(line)
```

# Context Managers

# Resource Management

- Proper management of system resources such as files, locks, and connections is often a tricky problem when combined with exceptions.
- For example, a raised exception can cause control flow to bypass statements responsible for releasing critical resources such as a lock.
- The `with` statement allows a series of statements to execute in a runtime context and guarantees to run the resource cleanup when control jumps out of this context.

# with Statement Example

```python
# Example 1
with open("debuglog","a") as f:
      f.write("Debugging\n")
      # some statements come here
      f.write("Done\n")
# file is closed when we come out of the with block

# Example 2
import threading
lock = threading.Lock()
with lock:
      # Critical section
      # some statements
      # End critical section
# lock is released when we come here
```

# Supporting contexts in your class

- You can support the context management using `with` statement for user defined classes by defining `__enter__()` and `__exit__()` methods.
- `__enter__()` is executed when the `with` block is entered. You can acquire resources in this method.
- `__exit__()` is executed when control jumps out of the `with` block. You can release resources in this method.

# Supporting contexts in your class ...2

```python
class ListTransaction(object):
    def __init__(self,thelist):
        self.thelist = thelist
    def __enter__(self):
        self.workingcopy = list(self.thelist)
        return self.workingcopy
    def __exit__(self,type,value,tb):
        if type is None:
            self.thelist[:] = self.workingcopy
        return False
```

# Classes with Context - Example usage

```python
items = [1,2,3]
with ListTransaction(items) as working:
    working.append(4)
    working.append(5)
print(items)       # Produces [1,2,3,4,5]

try:
    with ListTransaction(items) as working:
        working.append(6)
        working.append(7)
        raise RuntimeError("Forcing an exception")
except RuntimeError:
    pass
print(items)   # Produces [1,2,3,4,5].  6 & 7 are discarded
```

# Decorators

# Definition of Decorators

- A decorator is a function whose primary purpose is to wrap another function or class.

- The primary purpose of this wrapping is to transparently alter or enhance the behavior of the object being wrapped.

```python
@trace
def square(x):
     return x*x

# This is equivalent to:
def square(x):
    return x*x
square = trace(square)
```

# Tracing example

```python
enable_tracing = True
if enable_tracing:
    debug_log = open("debug.log","w")

def trace(func):
    if enable_tracing:
        def callf(*args,**kwargs):
            debug_log.write("Calling %s: %s, %s\n" %
                            (func.__name__, args, kwargs))
            r = func(*args,**kwargs)
            debug_log.write("%s returned %s\n" % (func.__name, r))
            return r
        return callf
    else:
        return func
```

# Tracing example

```
@trace
def square(x):
     return x**x
```

# Another Example - Memoization

```python
def cache(func):
    results = {} # dict acting as a cache
    def wrap(*args):
        key = tuple(args) # make it immutable
        if key in results:
            print("Returning pre-computed results!")
            result = results[key]
        else:
            result = func(*args)
            print("Computing a fresh result.")
            results[key] = result
        return result
    return wrap
```

```python
@cache
def heavy(a, b, c): return (a**b)**c
heavy(3, 4, 5) # fresh computation
heavy(7, 4, 6) # fresh computation
heavy(3, 4, 5) # returns stored result
heavy(7, 6, 4) # fresh computation
```

# Using Multiple Decorators

- When multiple decorators are used, each one should apply in their own line. For example,

```
@trace
@cache
def grok(x):
    pass
```

- This is equivalent to

```
def grok(x):
    pass
grok = trace(cache(grok))
```

# Passing Arguments to Decorators

– A decorator can also accept arguments. For example,

```
@eventhandler('BUTTON')
def handle_button(msg):
    ...
@eventhandler('RESET')
def handle_reset(msg):
    ...

# Event handler decorator
event_handlers = { }
def eventhandler(event):
    def register_function(f):
        event_handlers[event] = f
        return f
    return register_function
```

- A decorator can also be defined for classes, in which case the decorator should return a wrapped class.

# The End

Visit `https://www.python.org`

## Thank You