INFO-H-600 - Computing foundations of data sciences

TP 6
Introduction to Python
Advanced Python constructions

Université libre de Bruxelles
École polytechnique de Bruxelles

2020-2021

List comprehension

Suppose you have a collection of real numbers and you want a list of their absolute values

```
data = [5, -5, 3.6, -7.2, 9]
# result = [5, 5, 3.6, 7.2, 9]
```

- classic way :

```
result = []
for element in data:
    result.append(abs(element))
```

- using list comprehension :

```
result = [abs(element) for element in data]
```

Syntax :

```
[<the_expression> for <the_element> in <the_iterable>]
```

List comprehension

You can also filter elements easily

Suppose you want the square root of all positive numbers of your list :

- classic way :

```
result = []
for element in data:
    if element > 0:
        result.append(sqrt(element))
```

- using list comprehension :

```
result = [sqrt(element) for element in data if element > 0]
```

Syntax :

```
[<expression> for <element> in <iterable> if <condition>]
```

Positional arguments

- Until now, we have used functions with positional arguments :

```
def ratio(x, y):
    return x/y

x = 5
y = 4

print(ratio(y, x))      # 0.8   (not 1.25!)
```

Default arguments

- Another example :

```
def greet(name, msg):
    print(msg + ' ' + name)

greet("John", "Hello")
```

- If we try to use it without its arguments :

```
greet("John")     # only one argument
TypeError: greet() missing 1 required positional argument: 'msg'
```

Default arguments

- We can use default arguments :

```
def greet(name, msg="Good morning"):
    print(msg + ' ' + name)

greet("John", "Hi")   # Hi John
greet("John")         # Good morning John
```

Keyword arguments

- You can call a function by indicating to which argument your values correspond with keyword arguments :

```
 # 2 keyword arguments
greet(name = "John", msg = "Hello")

# 2 keyword arguments (out of order)
greet(msg = "Hello", name = "John")

# 1 positional, 1 keyword argument
greet("John", msg = "Hello")
```

- Be careful, not everything is possible :

```
greet(name="John","Hello") # error
```

- If you use positional arguments, they must be at the beginning !

lambda

- A lambda function is a small anonymous function
- can take any number of arguments, but can only have one expression

Syntax :

```
lambda arguments : expression
```

Example :

```
a = lambda x, y: str(x) *y
a(5,3)              #"555"
```

lambda : example

- Can be used to create a function . . . to create functions :

```
def my_multiplier(n):
  return lambda a : a * n

my_doubler = my_multiplier(2)
my_tripler = my_multiplier(3)

print(my_doubler(11))    # 22
print(my_tripler(11))    # 33
```

lambda and default arguments example

- The sort() method sorts the list ascending by default

```
list.sort(reverse=True, key=None)
```

- By default python will use a “classical” sorting algorithm and will sort elements of the list in increasing order

```
l = [1, 3, 2]
l.sort()
print(l)          # [1, 2, 3]
```

lambda and default arguments example

- Sort the following lists of tuple according to the value of its second element :

```
l = [(1, 3) , (5, 6), (2, 1)]
l.sort(key=lambda x: x[1])
print(l)          # [(2, 1), (1, 3), (5, 6)]
```

- All elements are evaluated according to the value they return when provided to the “key” function

Iterators

- Iterators are objects that can be iterated over

```
cities = ["Paris",
          "Frankfurt",
          "Amsterdam"]

for location in cities:
    print(location)
```

```
city_iterator = iter(cities)

while city_iterator:
    try:
        location = next(city_iterator)
        print(location)
    except StopIteration:
        break
```

- we can simulate for loops using a while loop

Implementing an Iterator

```
class MyIterator():

    def __init__(self, iterable):
        self.iterable = iterable
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):

        while self.i < len(self.iterable):
            obj = self.iterable[self.i]
            self.i += 1
            return obj
        raise StopIteration


x = MyIterator("abc")

for i in x:
    print(i)
# a, b, c
```

Generator and Generator Functions

- Generator is a function which creates a generator object
  - can be differentiated by a normal function thanks to the `yield` keyword

```
def city_generator():
    # any code
    yield("Bruxelles")
    # any code
    yield("London")

city = city_generator()  # generator object

# next() start or resume the generator object until the next yield
print(next(city)) # Bruxelles
print(next(city)) # London
```

Generator example

```
def count(firstval=0, step=1, stop=5):
    x = firstval
    while x < stop:
        yield x
        x += step


counter = count() # count will start with 0
for i in counter:
    print(i, end=", ")
# 0, 1, 2, 3, 4,


start_value = 2.1
step_value = 0.3
counter = count(start_value, step_value)
for i in counter:
    print(f"{i:2.2f}", end=", ")
# 2.10, 2.40, 2.70, 3.00, 3.30, 3.60, 3.90, 4.20, 4.50, 4.80,
```

Generator return statement

- The iterator will finish when the generator body has been completely executed or if a return statement without a value is encountered.

```
def myrange(n):
    number = 0
    while True:
        if (number > n):
            return
        yield number
        number += 1

f = myrange(5)
for x in f:
    print(x)
```

Difference between a generator and iterator

- iterator more general concept : any object whose class has a `__next__` method and an `__iter__` method that does return self.

- generator : object that is built by calling a function that has one or more yield expressions and is an object that meets the previous paragraph's definition of an iterator.
  Every generator is an iterator, but not vice versa.

```
def squares(start, stop):
    for i in range(start, stop):
        yield i * i

generator = squares(a, b)
```

Difference between a generator and iterator

```
class Squares(object):
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
    def __iter__(self): return self
    def __next__(self):
        if self.start >= self.stop:
            raise StopIteration
        current = self.start * self.start
        self.start += 1
        return current

iterator = Squares(a, b)
```

- easily add methods

```
    def current(self):
            return self.start
```

sources

- https://code.tutsplus.com/fr/tutorials/list-comprehensions-in-python--cms-26836
- https://www.programiz.com/python-programming/function-argument
- https://www.w3schools.com/python/python_lambda.asp
- https://www.python-course.eu/python3_generators.php
- https://stackoverflow.com/questions/2776829/difference-between-pythons-generators-and-iterat