PRG – PROGRAMMING ESSENTIALS

Lecture 12 – Introduction to Advanced Concepts

Milan Nemy

Czech Technical University in Prague,
Faculty of Electrical Engineering, Dept. of Cybernetics

https://beat.ciirc.cvut.cz/people/milan-nemy/

milan.nemy@cvut.cz

REFERENCES

- Lambda functions https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions
- List comprehensions https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions LICENSE
- Map – Filter – Reduce http://book.pythontips.com/en/latest/map_filter.html# LICENSE
- Iterators & Generators **http://book.pythontips.com/en/latest/generators.html LICENSE**
- Itertools by example https://realpython.com/python-itertools/

LAMBDA FUNCTIONS

- Small **anonymous** functions can be created with the **lambda** keyword

- EXAMPLE: *This function returns the sum of its two arguments*:

lambda a, b: a+b

- Lambda functions can be used wherever function objects are required
- Syntactically restricted to a **single expression**
- Like nested function definitions, lambda functions can reference variables from the **containing scope**

Source https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions LICENSE

LAMBDA FUNCTIONS

Lambdas are one line functions. They are also known as anonymous functions in some other languages. You might want to use lambdas when you don't want to use a function twice in a program. They are just like normal functions and even behave like them.

Blueprint

```
lambda argument: manipulate(argument)
```

Example

```
add = lambda x, y: x + y

print(add(3, 5))
# Output: 8
```

Source http://book.pythontips.com/en/latest/lambdas.html LICENSE

LAMBDA FUNCTIONS

Lambdas are one line functions. They are also known as anonymous functions in some other languages. You might want to use lambdas when you don't want to use a function twice in a program. They are just like normal functions and even behave like them.

Blueprint

```
lambda argument: manipulate(argument)
```

List sorting

```
a = [(1, 2), (4, 1), (9, 10), (13, -3)]
a.sort(key=lambda x: x[1])

print(a)
# Output: [(13, -3), (4, 1), (1, 2), (9, 10)]
```

Source http://book.pythontips.com/en/latest/lambdas.html LICENSE

LAMBDA FUNCTIONS

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Source https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions LICENSE

LIST COMPREHENSIONS

- The **list comprehensions** provide a concise way to **create lists**
- It consists of **[ ]** containing an expression followed by a **for** clause, then zero or more **for** or **if** clauses
- The **expressions can be anything** (any kind of objects in lists)
- The result will be a **new list** *created by evaluating the expression* in the context of the **for** and **if** clauses
- The list comprehension always returns a result **list**

- Common applications are to:
  Make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition

Source https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions LICENSE

LIST COMPREHENSIONS

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
In[2]: squares = []
  ...: for x in range(10):
  ...:     squares.append(x**2)
  ...:
In[3]: squares
Out[3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
In[2]: squares = [x**2 for x in range(10)]
  ...: squares
  ...:
Out[2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In[3]:
```

```
In[2]: vec = [-4, -2, 0, 2, 4]
  ...: # create a new list with the values doubled
  ...: result = [x*2 for x in vec]
  ...: result
  ...:
Out[2]: [-8, -4, 0, 4, 8]

In[3]:
```

Source https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions LICENSE

LIST COMPREHENSIONS

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
In[2]: vec = [-4, -2, 0, 2, 4]
   ...: # filter the list to exclude negative numbers
   ...: result = [x for x in vec if x >= 0]
   ...: result
   ...:
Out[2]: [0, 2, 4]
```

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
In[2]: vec = [-4, -2, 0, 2, 4]
   ...: # apply a function to all the elements
   ...: result = [abs(x) for x in vec]
   ...: result
   ...:
Out[2]: [4, 2, 0, 2, 4]
```

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
In[2]: # call a method on each element
   ...: freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
   ...: result = [weapon.strip() for weapon in freshfruit]
   ...: result
   ...:
Out[2]: ['banana', 'loganberry', 'passion fruit']
```

Source https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions LICENSE

LIST COMPREHENSIONS

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
In[2]: [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
Out[2]: [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

In[3]:
```

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
In[2]: [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
Out[2]: [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
In[3]: combs = []
   ...: for x in [1,2,3]:
   ...:     for y in [3,1,4]:
   ...:         if x != y:
   ...:             combs.append((x, y))
   ...:
   ...: combs
   ...:
Out[3]: [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Source https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions LICENSE

LIST COMPREHENSIONS

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
In[2]: # create a list of 2-tuples like (number, square)
   ...: result = [(x, x ** 2) for x in range(6)]
   ...: result
   ...:
Out[2]: [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

In[3]:
```

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
In[2]: # flatten a list using a listcomp with two 'for'
   ...: vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
   ...: result = [num for elem in vec for num in elem]
   ...: result
   ...:
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

In[3]:
```

Source https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions LICENSE

MAP – FILTER – REDUCE

`Map` applies a function to all the items in an input_list. Here is the blueprint:

Blueprint

```
map(function_to_apply, list_of_inputs)
```

Most of the times we want to pass all the list elements to a function one-by-one and then collect the output. For instance:

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
```

Source http://book.pythontips.com/en/latest/map_filter.html# LICENSE

MAP – FILTER – REDUCE

`Map` allows us to implement this in a much simpler and nicer way. Here you go:

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
```

Most of the times we use lambdas with `map` so I did the same. Instead of a list of inputs we can even have a list of functions!

Source http://book.pythontips.com/en/latest/map_filter.html# LICENSE

MAP – FILTER – REDUCE

```python
def multiply(x):
    return (x*x)
def add(x):
    return (x+x)

funcs = [multiply, add]
for i in range(5):
    value = list(map(lambda x: x(i), funcs))
    print(value)

# Output:
# [0, 0]
# [1, 2]
# [4, 4]
# [9, 6]
# [16, 8]
```

Source http://book.pythontips.com/en/latest/map_filter.html# LICENSE

MAP – FILTER – REDUCE

As the name suggests, `filter` creates a list of elements for which a function returns true. Here is a short and concise example:

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)

# Output: [-5, -4, -3, -2, -1]
```

The filter resembles a for loop but it is a builtin function and faster.

Source http://book.pythontips.com/en/latest/map_filter.html# LICENSE

MAP – FILTER – REDUCE

`Reduce` is a really useful function for performing some computation on a list and returning the result. It applies a rolling computation to sequential pairs of values in a list. For example, if you wanted to compute the product of a list of integers.

So the normal way you might go about doing this task in python is using a basic for loop:

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num

# product = 24
```

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])

# Output: 24
```

Source http://book.pythontips.com/en/latest/map_filter.html# LICENSE

GENERATORS

- Generators are *"functions"* that can be **paused and resumed** on the fly, returning an object that can be iterated over
- Unlike lists, generators are **lazy** and thus **produce items one at a time** and **only when asked**
- Generators are **much more memory efficient** when dealing with large datasets (*often the only way to handle large data*)
- One of the advanced python concepts is:
  How to create generator functions and expressions as well as why you would want to use them in the first place

Source https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

GENERATORS – TERMINOLOGY

3.1. Iterable

An `iterable` is any object in Python which has an `__iter__` or a `__getitem__` method defined which returns an **iterator** or can take indexes (You can read more about them here). In short an `iterable` is any object which can provide us with an **iterator**. So what is an **iterator**?

3.2. Iterator

An iterator is any object in Python which has a `next` (Python2) or `__next__` method defined. That's it. That's an iterator. Now let's understand **iteration**.

3.3. Iteration

In simple words it is the process of taking an item from something e.g a list. When we use a loop to loop over something it is called iteration. It is the name given to the process itself. Now as we have a basic understanding of these terms let's understand **generators**.

Source http://book.pythontips.com/en/latest/generators.html LICENSE

GENERATORS

Generators are iterators, but you can only iterate over them once. It's because they do not store all the values in memory, they generate the values on the fly. You use them by iterating over them, either with a 'for' loop or by passing them to any function or construct that iterates. Most of the time `generators` are implemented as functions. However, they do not `return` a value, they `yield` it. Here is a simple example of a `generator` function:

```
def generator_function():
    for i in range(10):
        yield i

for item in generator_function():
    print(item)

# Output: 0
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

Source http://book.pythontips.com/en/latest/generators.html LICENSE

GENERATORS

Here is an example `generator` which calculates fibonacci numbers:

```
# generator version
def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b
```

Now we can use it like this:

```
for x in fibon(1000000):
    print(x)
```

This way we would not have to worry about it using a lot of resources. However, if we would have implemented it like this:

```
def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result
```

Source http://book.pythontips.com/en/latest/generators.html LICENSE

GENERATORS

It would have used up all our resources while calculating a large input. We have discussed that we can iterate over `generators` only once but we haven't tested it. Before testing it you need to know about one more built-in function of Python, `next()`. It allows us to access the next element of a sequence. So let's test out our understanding:

```
def generator_function():
    for i in range(3):
        yield i

gen = generator_function()
print(next(gen))
# Output: 0
print(next(gen))
# Output: 1
print(next(gen))
# Output: 2
print(next(gen))
# Output: Traceback (most recent call last):
#           File "<stdin>", line 1, in <module>
#         StopIteration
```

Source http://book.pythontips.com/en/latest/generators.html LICENSE

GENERATORS

As we can see that after yielding all the values `next()` caused a `StopIteration` error. Basically this error informs us that all the values have been yielded. You might be wondering that why don't we get this error while using a `for` loop? Well the answer is simple. The `for` loop automatically catches this error and stops calling `next`. Do you know that a few built-in data types in Python also support iteration? Let's check it out:

```
my_string = "Yasoob"
next(my_string)
# Output: Traceback (most recent call last):
#     File "<stdin>", line 1, in <module>
#   TypeError: str object is not an iterator
```

Source http://book.pythontips.com/en/latest/generators.html LICENSE

GENERATORS

Well that's not what we expected. The error says that `str` is not an iterator. Well it's right! It's an iterable but not an iterator. This means that it supports iteration but we can't iterate over it directly. So how would we iterate over it? It's time to learn about one more built-in function, `iter`. It returns an `iterator` object from an iterable. While an `int` isn't an iterable, we can use it on string!

```
int_var = 1779
iter(int_var)
# Output: Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: 'int' object is not iterable
# This is because int is not iterable

my_string = "Yasoob"
my_iter = iter(my_string)
print(next(my_iter))
# Output: 'Y'
```

Now that is much better. I am sure that you loved learning about generators. Do bear it in mind that you can fully grasp this concept only when you use it. Make sure that you follow this pattern and use `generators` whenever they make sense to you. You won't be disappointed!

Source http://book.pythontips.com/en/latest/generators.html LICENSE

GENERATORS – SUMMARY

Generators:

A generator is a function that produces or yields a sequence of values using `yield` method.

Every `next()` method call on generator object(for ex: `f` as in below example) returned by generator function(for ex: `foo()` function in below example), generates next value in sequence.

When a generator function is called, it returns an generator object without even beginning execution of the function. When `next()` method is called for the first time, the function starts executing until it reaches yield statement which returns the yielded value. The yield keeps track of i.e. remembers last execution. And second `next()` call continues from previous value.

Source http://openbookproject.net/thinkcs/python/english3e/app_a.html

REFERENCES

This lecture re-uses selected parts of the OPEN BOOK PROJECT
Learning with Python 3 (RLE)
http://openbookproject.net/thinkcs/python/english3e/index.html
available under **GNU Free Documentation License Version 1.3**)

- Version date: October 2012
- by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers
  (based on 2nd edition by Jeffrey Elkner, Allen B. Downey, and Chris Meyers)
- Source repository is at https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle
- For offline use, download a zip file of the html or a pdf version
  from http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/

OTHER SOURCES USED:

- Lambda functions https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions
- List comprehensions https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions LICENSE
- Map – Filter – Reduce http://book.pythontips.com/en/latest/map_filter.html# LICENSE
- Iterators & Generators **http://book.pythontips.com/en/latest/generators.html LICENSE**
- Itertools by example https://realpython.com/python-itertools/

EXAMPLE – PRIME NUMBERS

TASK: Write a program to generate a list of all prime numbers less than 20

- Before starting it is important to note what a prime number is:
  - *A prime number has to be a positive integer*
  - *Divisible by exactly 2 integers (1 and itself)*
  - *1 is not a prime number*
- While there are many different ways to solve this problem, here are a few different approaches

SOURCE: https://hackernoon.com/prime-numbers-using-python-824ff4b3ea19

source https://hackernoon.com/prime-numbers-using-python-824ff4b3ea19

EXAMPLE – PRIME NUMBERS

```python
# Approach 1: ForLoops
# Initialize a list
primes = []
for possiblePrime in range(2, 21):

    # Assume number is prime until shown it is not.
    isPrime = True
    for num in range(2, possiblePrime):
        if possiblePrime % num == 0:
            isPrime = False

    if isPrime:
        primes.append(possiblePrime)

```

- Example of a solution

source https://hackernoon.com/prime-numbers-using-python-824ff4b3ea19

EXAMPLE – PRIME NUMBERS

```python
# Approach 1: ForLoops
# Initialize a list
primes = []
for possiblePrime in range(2, 21):

    # Assume number is prime until shown it is not.
    isPrime = True
    for num in range(2, possiblePrime):
        if possiblePrime % num == 0:
            isPrime = False

    if isPrime:
        primes.append(possiblePrime)

```

- Approach 1: notice that as soon **isPrime** is False, it is inefficient to keep on iterating. It would be more efficient to **exit out of the loop**.

source https://hackernoon.com/prime-numbers-using-python-824ff4b3ea19

EXAMPLE – PRIME NUMBERS

```
15
16  # Approach2: For Loops with Break
17  # Initialize a list
18   primes = []
19  for possiblePrime in range(2, 21):
20
21      # Assume number is prime until shown it is not.
22      isPrime = True
23      for num in range(2, possiblePrime):
24          if possiblePrime % num == 0:
25              isPrime = False
26              break
27
28      if isPrime:
29          primes.append(possiblePrime)
30
```

- Approach 2 is more efficient than approach 1 because as soon as you find a given number isn't a prime number you can exit out of loop using break.

source https://hackernoon.com/prime-numbers-using-python-824ff4b3ea19

EXAMPLE – PRIME NUMBERS

```python

# Approach 3: For Loop, Break, and Square Root
# Initialize a list
primes = []
for possiblePrime in range(2, 21):

    # Assume number is prime until shown it is not.
    isPrime = True
    for num in range(2, int(possiblePrime ** 0.5) + 1):
        if possiblePrime % num == 0:
            isPrime = False
            break

    if isPrime:
        primes.append(possiblePrime)

```

- Approach 3: is similar to approach 2 except the inner range function. Notice that the inner range function is now:
  range(2, int(possiblePrime ** 0.5) + 1)

source https://hackernoon.com/prime-numbers-using-python-824ff4b3ea19

EXAMPLE – PRIME NUMBERS

- We use the properties of **composite numbers**
- Composite number is a **positive** number **greater than 1** that is **not prime** (which has factors other than 1 and itself)
- Every composite number has a factor **less than or equal to its square root** (proof here).

- **EXAMPLE**: *Factors of 15 below; the factors in red are just the reverses of the green factors so by the commutative property of multiplication 3 x 5 = 5 x 3 we just need to include the "green" pairs to be sure that we have all the factors.*

Factors of 15				
Factor 1	1	3	5	15
Factor 2	15	5	3	1

source https://hackernoon.com/prime-numbers-using-python-824ff4b3ea19

EXAMPLE – PRIME NUMBERS

```
55  import timeit
56
57  # Approach 1: Execution time
58  print(timeit.timeit('approach1(500)', globals=globals(), number=100000))
59  # Approach 2: Execution time
60  print(timeit.timeit('approach2(500)', globals=globals(), number=100000))
61  # Approach 3: Execution time
62  print(timeit.timeit('approach3(500)', globals=globals(), number=100000))
63
```

- Evaluating performance

- REFERENCE: https://hackernoon.com/prime-numbers-using-python-824ff4b3ea19

source https://hackernoon.com/prime-numbers-using-python-824ff4b3ea19