



# Lecture 16: Functional Programming & Other In-Built Functions

## IN628: Programming 4

### Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

# LECTURE 01: PYTHON 1 RECAP

- ▶ Object-oriented programming principles
- ▶ Basic data structures
- ▶ Comprehensions

## LECTURE 02: PYTHON 2 TOPICS

- ▶ Functional programming
- ▶ Context managers
- ▶ Other in-built functions

# FUNCTIONAL PROGRAMMING

- ▶ Lambda
- ▶ Map
- ▶ Filter
- ▶ Reduce
- ▶ Iterators
- ▶ Generators

# LAMBDA

- ▶ Lambda expressions/forms
- ▶ Used to create anonymous functions
- ▶ The expression yields a function object
- ▶ The unnamed object behaves like a function object

```
lambda parameters: expression
```

```
def lambda(parameters):  
    return expression
```

# MAP

- ▶ Returns a map object/iterator
- ▶ Applies a given function to each item in a given iterable
- ▶ Yields the results

```
def power_of_three(x):  
    return x ** 3  
  
nums = [1, 2, 3, 4, 5]  
pow_three = map(power_of_three, nums)  
print(pow_three) # <map object at 0x10c28eb50>  
print(list(pow_three)) # [1, 8, 27, 64, 125]  
  
pow_three = map(lambda x: x ** 3, nums)  
print(pow_three) # <map object at 0x10c28eb50>  
print(list(pow_three)) # [1, 8, 27, 64, 125]
```

# FILTER

- Creates an iterator from each item in a given iterable where a given function returns true

```
def even_numbers(x):  
    return x % 2 == 0  
  
nums = [1, 2, 3, 4, 5]  
even_nums = filter(even_numbers, nums)  
print(even_nums) # <filter object at 0x10c34e850>  
print(list(even_nums)) # [2, 4]  
  
even_nums = filter(lambda x: x % 2 == 0, nums)  
print(even_nums) # <filter object at 0x10c34e850>  
print(list(even_nums)) # [2, 4]
```

# REDUCE

- ▶ functools module
- ▶ Applies a function of two arguments cumulatively to each item in a given iterable
- ▶ Reduces a given iterable to a single value
- ▶ The left argument is the accumulated values
- ▶ The right argument is the update value from the given iterable

```
from functools import reduce

def sum_numbers(x, y):
    return x + y

nums = [1, 2, 3, 4, 5]
sum_nums = reduce(sum_numbers, nums)
print(sum_nums) # 15

sum_nums = reduce(lambda x, y: x + y, nums)
print(sum_nums) # 15
```



# ITERATORS

- ▶ An object representing a stream of data
- ▶ This object returns the data one item at a time
- ▶ Must support the `__next__()` method
- ▶ If there is no more items in the stream, the `__next__()` method must raise the `StopIteration` exception
- ▶ Iterators don't have to be finite

# ITERATORS

- ▶ `iter()` & `__iter__()`
- ▶ `next()` & `__next__()`

```
numbers = [1, 2, 3, 4, 5]
numbers_iter = iter(numbers)
print(next(numbers_iter)) # 1
print(next(numbers_iter)) # 2
print(next(numbers_iter)) # 3
print(next(numbers_iter)) # 4
print(next(numbers_iter)) # 5
print(next(numbers_iter)) # StopIteration:
```

# ITERATORS

## ► Iterator class

```
class PowerOfThree:
    def __init__(self, min_num, max_num):
        self.min_num = min_num
        self.max_num = max_num

    def __iter__(self):
        return self

    def __next__(self):
        if self.min_num <= self.max_num:
            result = 3 ** self.min_num
            self.min_num += 1
            return result
        else:
            raise StopIteration

def main():
    pow_three = PowerOfThree(0, 3)
    pow_three_iter = pow_three.__iter__()
    print(pow_three_iter.__next__())
    print(pow_three_iter.__next__())
    print(pow_three_iter.__next__())
    print(pow_three_iter.__next__())
    print(pow_three_iter.__next__())

if __name__ == '__main__':
    main()
    # 1
    # 3
    # 9
    # 27
    # StopIteration:
```

# GENERATORS

- ▶ Simplifies the task of writing iterators
- ▶ Returns an iterator that returns a stream of data
- ▶ Any function containing the yield keyword is a generator function
- ▶ The big difference between yield & a return statement:
  - ▶ The generator's state of execution is suspended
  - ▶ Local variables are preserved
- ▶ The function will resume executing on the next call to the generator's `__next__()` method

```
def power_of_three(max_num):  
    min_num = 0  
    while min_num <= max_num:  
        yield 3 ** min_num  
        min_num += 1  
  
pow_three = power_of_three(3)  
print(next(pow_three)) # 1  
print(next(pow_three)) # 3  
print(next(pow_three)) # 9  
print(next(pow_three)) # 27  
print(next(pow_three)) # StopIteration:
```

# CONTEXT MANAGERS

- ▶ An object that defines the runtime context to be established when executing a with statement
- ▶ Handles the entry to & exit from the runtime context

```
with open('hello-world.txt', 'w') as f:  
    f.write('Hello_World')  
  
f = open('hello-world.txt', 'w')  
try:  
    f.write('Hello_World')  
finally:  
    f.close()
```

# CONTEXT MANAGERS: CLASS

## ► Context manager class

```
class File:
    def __init__(self, filename, mode):
        self.file_obj = open(filename, mode)

    def __enter__(self):
        return self.file_obj

    def __exit__(self, type, value, traceback):
        self.file_obj.close()

def main():
    with File('hello-world.txt', 'r') as f:
        contents = f.read()
        print(contents)

if __name__ == '__main__':
    main() # Hello World
```

# CONTEXT MANAGERS: GENERATOR

- ▶ contextlib module
- ▶ @contextmanager

```
from contextlib import contextmanager

@contextmanager
def open_file(filename):
    f = open(filename, 'r')
    yield f
    f.close()

with open_file('hello-world.txt') as f:
    contents = f.read()
    print(contents)  # Hello World
```

## OTHER IN-BUILT FUNCTIONS

- ▶ Enumerate
- ▶ Reversed
- ▶ Slice
- ▶ Sorted
- ▶ Vars
- ▶ Zip



# ENUMERATE

- ▶ Returns an enumerate object
- ▶ The given iterable must be a sequence, a iterator or an object that supports iteration
- ▶ The `__next__()` method returned by the `enumerate()` function returns:
  - ▶ A tuple containing a count
  - ▶ The values obtained from iterating over the given iterable

```
first_names = ['Fran', 'Tosha', 'Margarito', 'Junie', 'Christel']
last_names = ['Piggott', 'Hurley', 'Kirkman', 'Purdy', 'Edmundson']
first_names_enumerate = enumerate(first_names)
last_names_enumerate = enumerate(first_names, start=1)
print(type(first_names_enumerate)) # <class 'enumerate'>
print(first_names_enumerate) # <enumerate object enumerate at 0x105e88450>
print(list(first_names_enumerate)) # [(0, 'Fran'), (1, 'Tosha'),
                                     # (2, 'Margarito'), (3, 'Junie'), (4, 'Christel')]
print(list(last_names_enumerate)) # [(1, 'Fran'), (2, 'Tosha'),
                                     # (3, 'Margarito'), (4, 'Junie'), (5, 'Christel')]

def enumerate(sequence, start=0):
    for item in sequence:
        yield start, item
        start += 1

print(list(enumerate(first_names))) # [(0, 'Fran'), (1, 'Tosha'),
                                     # (2, 'Margarito'), (3, 'Junie'), (4, 'Christel')]
```

# REVERSED

- ▶ Returns a reverse iterator object & the items of a given sequence in reverse order
- ▶ The given sequence must be an object which has a `__reversed__()` method or supports the sequence protocol

```
first_names = ['Fran', 'Tosha', 'Margarito', 'Junie', 'Christel']
first_names_reversed = reversed(first_names)
print(type(first_names_reversed)) # <class 'list_reverseiterator'>
print(first_names_reversed) # <list_reverseiterator object at 0x105dcf4d0>
print(list(first_names_reversed)) # ['Christel', 'Junie', 'Margarito', 'Tosha', 'Fran']
```

# SLICE

- Returns a slice object representing the set of indices specified by `range(start, stop, step)`
- Used to slice an object which supports the sequence protocol

```
first_names = ['Fran', 'Tosha', 'Margarito', 'Junie', 'Christel']
slice_start = slice(2)
slice_start.end = slice(2, 5)
slice_start.end.step = slice(2, 5, 2)
print(type(slice_start)) # <class 'slice'>
print(first_names[slice_start]) # ['Fran', 'Tosha']
print(first_names[slice_start.end]) # ['Margarito', 'Junie', 'Christel']
print(first_names[slice_start.end.step]) # ['Margarito', 'Christel']
```

# SORTED

- ▶ Sorts & returns the items of a given iterable in a specific order - ascending (default) or descending
- ▶ Two optional arguments (key & reverse) which must be specified as keyword arguments

```
first_names = ['Fran', 'Tosha', 'Margarito', 'Junie', 'Christel']
last_names = ['Piggott', 'Hurley', 'Kirkman', 'Purdy', 'Edmundson']
first_names_sorted_asc = sorted(first_names)
last_names_sorted_desc = sorted(last_names, reverse=True)
print(type(first_names_sorted_asc)) # <class 'list'>
print(first_names_sorted_asc) # ['Christel', 'Fran', 'Junie', 'Margarito', 'Tosha']
print(last_names_sorted_desc) # ['Purdy', 'Piggott', 'Kirkman', 'Hurley', 'Edmundson']
```

# VARS

- Returns the `--dict--` attribute for a module, class, instance or an object with a `--dict` attribute--

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []

    def add_trick(self, trick):
        self.tricks.append(trick)

def main():
    dog = Dog('Fido')
    dog.add_trick('roll_over')
    dog.add_trick('play_dead')
    print(type(vars(dog)))
    print(vars(dog))

if __name__ == '__main__':
    main()
    # <class 'dict'>
    # {'name': 'chihuahua', 'tricks': ['roll over', 'play dead']}
```

# ZIP

- Returns an iterator of tuples where the *i-th* tuple contains the *i-th* element from each of the given sequences or iterables
- The iterator stops when the shortest given sequence or iterable is exhausted

```
first_names = ['Fran', 'Tosha', 'Margarito', 'Junie', 'Christel']
last_names = ['Piggott', 'Hurley', 'Kirkman', 'Purdy', 'Edmundson']
first_last_names_zip = zip(first_names, last_names)
print(type(first_last_names_zip)) # <class 'zip'>
print(first_last_names_zip) # <zip object at 0x105df1d20>
print(list(first_last_names_zip)) # [('Fran', 'Piggott'), ('Tosha', 'Hurley'),
                                   # ('Margarito', 'Kirkman'), ('Junie', 'Purdy'),
                                   # ('Christel', 'Edmundson')]
```

## ► Quick calculations

```
months = ['Jan', 'Feb', 'Mar', 'Apr']
revenue_per_month = [44611.00, 47976.00, 47535.00, 45383.00]
cost_per_month = [46893.00, 43157.00, 41164.00, 40761.00]
calculations = zip(months, revenue_per_month, cost_per_month)
for m, r, c in calculations:
    profit = r - c
    print(f'Profit for {m}: {profit}') # Profit for Jan: -2282.0
                                     # Profit for Feb: 4819.0
                                     # Profit for Mar: 6371.0
                                     # Profit for Apr: 4622.0
```

# ZIP: UNPACKING

- Unpacking iterables (single asterisk \*)
- Unpacking dictionaries (double asterisk \*\*)

```
months = ['Jan', 'Feb', 'Mar', 'Apr']
revenue_per_month = [44611.00, 47976.00, 47535.00, 45383.00]
cost_per_month = [46893.00, 43157.00, 41164.00, 40761.00]
calculations = zip(months, revenue_per_month, cost_per_month)
unpacking_calculations = zip(*calculations)
print(list(unpacking_calculations)) # [('Jan', 'Feb', 'Mar', 'Apr'),
                                     # (44611.0, 47976.0, 47535.0, 45383.0),
                                     # (46893.0, 43157.0, 41164.0, 40761.0)]
```



# DEL

- ▶ Deletion of a target
  - ▶ Each target from left to right is recursively deleted
- ▶ Deletion of a name
  - ▶ The name's binding is removed from the local or global namespace
  - ▶ A `NameError` exception will be raised, if the name is unbound

```
x = 10
print(x) # 10
del x
print(x) # NameError: name 'x' is not defined
```