# Python 3: Functional Programming

**IN608: Intermediate Application Development Concepts**

**Kaiako: Tom Clark & Grayson Orr**

# Last Session's Content

- More abstract data types
  - List as a stack
  - Stack class
  - List as a queue
  - Queue class
  - Circular queue

# Today’s Content

- Comprehension
  - List
  - Set
  - Dictionary
- First-class functions
- Lambda expression
- Map
- Filter
- Reduce
- Iterator
- Generator

# Comprehension

# Comprehension

- Succinct way of creating a list, set or dictionary
- A comprehension consists of the following components:
  - Expression (optional)
  - Variable
  - Input sequence
  - Predicate (optional)

```
[expression for variable input sequence predicate]
```

# List Comprehension

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

```
string = '123 Hi 456'
nums = []
for s in string:
    if s.isdigit():
        nums.append(int(s))
print(nums)

# is equivalent to

string = '123 Hi 456'
nums = [int(s) for s in string if s.isdigit()]
print(nums)
```

# Set Comprehension

```
class Cat:
    def __init__(self, breed, is_active):
        self.breed = breed
        self.is_active = is_active

def main():
    cats = [
        Cat('Birman', True),
        Cat('Birman', True),
        Cat('Maine Coon', False),
        Cat('Persian', False),
        Cat('Ragdoll', False),
        Cat('Siamese', True)
    ]
    active_cats = {c.breed for c in cats if c.is_active}
    print(active_cats)

if __name__ == '__main__':
    main() # {‘Birman’, ‘Siamese’}
```

# Dictionary Comprehension

```
fruit_price = {'apple': 0.89, 'banana': 0.75, 'orange': 0.60, 'pineapple': 3.50}
double_fruit_price = {}
for (k, v) in fruit_price.items():
    double_fruit_price[k] = v * 2
print(double_fruit_price) # {'apple': 1.78, 'banana': 1.5, 'orange': 1.2, 'pineapple': 7.0}

# is equivalent to

fruit_price = {'apple': 0.89, 'banana': 0.75, 'orange': 0.60, 'pineapple': 3.50}
double_fruit_price = {k: v * 2 for (k, v) in fruit_price.items()}
print(double_fruit_price) # {'apple': 1.78, 'banana': 1.5, 'orange': 1.2, 'pineapple': 7.0}
```

# Programming Activity (30 Minutes)

# Programming Activity

- Checkout to master - `git checkout master`
- Create a new branch called 03-practical - `git checkout -b 03-practical`
- Copy 03-practical.ipynb from the course materials repository into your practicals repository
- Open up the Anaconda Prompt (it should be install on all lab computers) & `cd` to your practicals repository
- Run the following command: `jupyter notebook`

# Programming Activity

- Please open 03-practical.ipynb
- Please **ONLY** answer questions 1-3
- We will go through the solutions after 30 minutes

# Solutions

# First-Class Functions

# First-Class Functions

- Functions in Python are first-class objects
- Assign a function as a value to a variable
- Pass a function as an argument to another function
- Return a function from another function

# First-Class Functions

```
def double(x):
    return x * 2

times_two = double
print(times_two(4)) # 8

def f(func, arg):
    return func(arg)

print(f(times_two, 4)) # 8

def multiply(factor):
    def f(x):
        return factor * x
    return f

three = multiply(2)
print(three(4)) # 8
```

# Lambda Expression

# Lambda Expression

- Lambda expression or lambda form
- Used to create an anonymous/unnamed function
- Generally used as an argument to a higher-order function
- `lambda parameters: expression` - yields a function object
- Resource: https://docs.python.org/3/reference/expressions.html#lambda

```
lambda parameters: expression

add = lambda x, y: x + y
print(add(5, 5)) # 10
```

# Map

# Map

- `map(function, iterable)`
- Returns an iterator which applies *`function`* to elements of *`iterable`*, yielding the results
- Resource: https://docs.python.org/3/library/functions.html#map

```
# Named function
def power_of_three(x):
    return x ** 3

# Anonymous function
power_of_three = lambda x: x ** 3

nums = [x for x in range(1, 11)]
pow_of_three_nums = map(power_of_three, nums)
print(type(pow_of_three_nums)) # <class 'map'>
print(list(pow_of_three_nums)) # [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

# Filter

# Filter

- `filter(function, iterable)`
- Constructs an iterator from elements of *`iterable`* for which a *`function`* returns true
- Resource: https://docs.python.org/3/library/functions.html#filter

```
# Named function
def is_even(x):
    return x % 2 == 0

# Anonymous function
is_even = lambda x: x % 2 == 0

nums = [x for x in range(1, 11)]
even_nums = filter(is_even, nums)
print(type(even_nums)) # <class filter>
print(list(even_nums)) # [2, 4, 6, 8, 10]
```

# Reduce

# Reduce

- `functools` module
- `reduce(function, iterable)`
- Applies *`function`* of two arguments cumulatively to elements of *`iterable`* from left to right, reducing the iterable to a single value
- Resource: https://docs.python.org/3/library/functools.html#functools.reduce

```
from functools import reduce

# Named function
def add(x, y):
    return x + y

# Anonymous function
add = lambda x, y: x + y

nums = [x for x in range(1, 11)]
sum_nums = reduce(add, nums)
print(sum_nums) # 55
```

# Iterator

# Iterator

- `iter(object)` - returns an iterator object. *`object`* must be a collection which supports the iteration or sequence protocol
- `next(iterator)` - retrieves the next item from the *`iterator`* by calling its `__next__()` method
- Resources:
  - https://docs.python.org/3/library/functions.html#iter
  - https://docs.python.org/3/library/functions.html#next

```
pow_of_three_nums = [x ** 3 for x in range(1, 6)]
pow_of_three_iter = iter(pow_of_three_nums)
print(type(pow_of_three_iter)) # <class 'list_iterator'>
print(next(pow_of_three_iter)) # 1
print(next(pow_of_three_iter)) # 8
print(next(pow_of_three_iter)) # 27
print(next(pow_of_three_iter)) # 64
print(next(pow_of_three_iter)) # 125
print(next(pow_of_three_iter)) # StopIteration
```

# Iterator Class

- Iterator class

```
class PowerOfThree:
    def __init__(self, min, max):
        self.min = min
        self.max = max

    def __iter__(self):
        return self

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

def main():
    pow_of_three = PowerOfThree(1, 5)
    pow_of_three_iter = iter(pow_of_three)
    print(next(pow_of_three_iter)) # 1
    print(next(pow_of_three_iter)) # 8
    print(next(pow_of_three_iter)) # 27
    print(next(pow_of_three_iter)) # 64
    print(next(pow_of_three_iter)) # 125
    print(next(pow_of_three_iter)) # StopIteration

if __name__ == '__main__':
    main()
```

# Generator

# Generator

- Returns a generator iterator
- Contains a `yield` for producing values that can be retrieved one at a time with the `next()` function

```
def power_of_three(min, max):
    while min <= max:
        yield min ** 3
        min += 1

pow_of_three = power_of_three(1, 5)
print(type(pow_of_three)) # <class 'generator'>
print(next(pow_of_three)) # 1
print(next(pow_of_three)) # 8
print(next(pow_of_three)) # 27
print(next(pow_of_three)) # 64
print(next(pow_of_three)) # 125
print(next(pow_of_three)) # StopIteration
```