Functional Programming

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Introduction

Most of us are primarily familiar with object-oriented programming. It's principal abstractions are around classes and objects. Objects are frequently used to maintain state. Python is an object-oriented language.

There are completely different types of programming and programming languages. One of them that is used today is *functional programming*. Examples of functional programming languages include

1. Lisp
2. Haskell
3. Javascript

Python is not a functional language, but it does support some functional programming features.

Key idea: Pure Functions

A *pure function* has two properties.

1. It always returns the same result when given the same arguments;
2. It has no side effects.

Here is an example of a pure function.

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

Functional programming relies on pure functions.

Sidebar: Impure Functions

This function is not pure:

```
random.randint(0, 100)
```

Given the same arguments (0, 100) it will not always return the same result.

Neither is this one

```
l = [1, 2, 3]
l.pop()
```

Besides returning the last item on the list, it also **changes** the list by removing the last item. This is a side effect.

Key Idea: First Class Functions

Functions in Python are first-class values.

- We can assign a function's value to a variable.
- Functions can be passed as arguments to other functions.
- Functions can be returned from other functions.

First class functions are not required for functional programming, but they make it easier.

First Class Functions: Examples

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

times_two = double # <- notice the lack of brackets
print(times_two(2)) # 4

def apply_twice(fn, arg):
    return fn(arg), fn(arg)

apply_twice(double, 2)  # returns 4, 4

def make_multiplier(factor):
    def mult(x):
        return x * factor
    return mult

times_two = make_multiplier(2)
```

Sidebar: Closures

Hey, wait...

```
def make_multiplier(factor):
 def mult(x):
   return x * factor
 return mult
```

Notice how our function `mult()` uses the local variable `factor` that goes out of scope when the function `make_multiplier()` exits?
The function encloses its lexical scope at the time it is defined. We call this a *closure*.

One More Idea: Lambdas

Sometimes we need a small function in a very specific context.
The following are equivalent

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

double = lambda x: x * 2   # <- Note the lack of "return".
```

Typically we use lambdas when we need a function as an argument to another function

```
foo(lambda x: x * 6, [2, 7, 'cat'])
```

Another name for lambda is *anonymous function*.
N.B.: In Python lambdas are limited to only one expression.

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `05-practical` in your practicals repo.
3. Add a subdirectory, `05-practical` and copy `04-practical.ipynb` from the class materials into it.
4. Open a shell, cd to this directory, and run `jupyter notebook` to open the notebook. Complete the first questions.
5. We will discuss results in 30ish minutes.

Map

```
map(function, iterable) -> iterator
```

Returns iterator which applies `function` to elements of `iterable`, yielding the results

```
nums = [1, 2, 3, 4, 5]
cubes = map(lambda x: x ** 3, nums)
print(cubes)  # [1, 8, 27, 64, 125]
```

Filter

`filter(function, iterable) -> iterator`
Returns iterator which yields elements of `iterable` for which `function` returns `True`.

```
def is_even(num):
    returns x % 2 == 1

nums = [1, 2, 3, 4, 5]
odds = filter(is_even, nums)
print(odds) # [1, 3, 5]
```

Reduce

```
reduce(function, iterable) -> value
```

Applies `function` of two arguments cumulatively to elements of iterable from left to right, reducing the iterable to a single value

From the `functools` module

```
from functools import reduce

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

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

Partial

`partial(function, *args) -> partial object`
Returns a function-like object (basically a function). This new function behaves like the original function with `*args` supplied to it.
From the `functools` module

```
from functools import partial

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

add_two = partial(add, 2)
# add_two is like add(2, x)
add_two(3)  # returns 5
```

List (and other) Comprehensions

The functionality of `map` and `filter` is combined in a *list comprehension*. This is why it’s rare to see `map` and `filter` in Python.

```
string = ’123 Hi 456’
nums = [int(s) for s in string if s.isdigit()]
print(nums)  # [1, 2, 3, 4, 5, 6]
```

This is equivalent to

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

There are also set and dictionary comprehensions that work in a similar way.