W02_04_Functional_Programming

September 12, 2022

1 Functional Programming

"Functional Programming" means programming with an emphasis on functions as the basic building blocks. It is particularly characterised by the use of higher-order functions and functions without side-effects, which we'll introduce below. In fact, we've already started doing functional programming. We'll start with the easiest concept, programming without side effects.

1.0.1 Functions without side effects

A *side effect* is something that a function does *other* than calculate and return a value. For example, printing to the terminal and writing to a file are side effects. Also, changing the values of an argument is a side effect. Example:

[2]:
```
def concat(a, b):
    a += b
    return a
a = [1, 2, 3]
b = [4, 5, 6]
c = concat(a, b)
print(a)
print(b)
print(c)
```

```
[1, 2, 3, 4, 5, 6]
[4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

As we see above, although the *return value* `c` is correct, this function has a side effect: it changes the first argument `a`. It's *often* better to avoid such effects. Programming without side-effects tends to allow for "pure", clean program design, and makes testing much easier.

[3]:
```
def concat(a, b):
    return a + b
a = [1, 2, 3]
b = [4, 5, 6]
c = concat(a, b)
print(a)
print(b)
print(c)
```

```
[1, 2, 3]
[4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

1.0.2 any and all

It is common to want to check whether all items in a list satisfy some *predicate*, e.g. to check whether all items are odd:

[12]:
```
flag = True
L = [17, 19, 23, 31, 32]
for x in L:
    if x % 2 == 0: # x even?
        flag = False
        break
print(flag)
```

```
False
```

In functional programming, we would try to improve this. First, we extract the predicate:

[10]:
```
# notice, no need for an if-else here!
def even(n): return n % 2 == 0
```

Next, Python provides a nice function `all` which reduces the whole thing to one line. The argument to `all` can be list, or another iterable type such as a generator comprehension. Note, we are not using a *higher-order* function here, as we are passing in Boolean values, not a function.

[11]:
```
all(not even(x) for x in L)
```

[11]:
```
False
```

1.0.3 map

"Of course someone has to write loops. It doesn't have to be you." – Jenny Bryan

Adapted from Wickham, *The Joy of Functional Programming*

The `map` function is a central example of functional programming. It takes two arguments, a function `f` and a list `L`, and returns a new list created by applying `f` to each element of `L`. For example:

[13]:
```
list(map(len, ["a", "cat", "and", "a", "dog"]))
```

[13]:
```
[1, 3, 3, 1, 3]
```

Note that we are passing in `len`, not `len()`. We are passing in a function, not the return value of a function.

`map` doesn't return a list: it returns an iterator, a bit like the generators we saw previously:

[14]:
```
map(len, ["a", "cat", "and", "a", "dog"])
```

[14]:
```
<map at 0x111bebc50>
```

That's why, if we just want to see the results immediately (as opposed to iterate over them), we just enclose the `map` in a `list`:

[15]:
```
list(map(len, ["a", "cat", "and", "a", "dog"]))
```

[15]:
```
[1, 3, 3, 1, 3]
```

`map` is an example of functional programming because our attention is on the transformation represented by `len` and the higher-order structure represented by `map`. We are not distracted with details of `for`-loops and initialisations.

Exercise: make your own implementation of `map` (but call it `mymap` to avoid overriding the builtin `map`) and show that it works using the example above.

`lambda` is common in combination with `map`, e.g.:

[16]:
```
list(map(lambda x: x**2, [4, 5, 6]))
```

[16]:
```
[16, 25, 36]
```

Exercise: calculate e^x for every value of $x \in [0.0, 0.1, 0.2, ...1.0]$. Use `range` to make a range of integers, then `lambda` and `map` to transform it.

Adapted from Wickham, *The Joy of Functional Programming*

So far, with `map` we've only seen functions `f` which take just *one* argument. What if we want to work with a function `f` which takes more than one? For example, what if we had lists of numbers `x` and `y` and we wanted to calculate `x * y`. Well, it turns out that this works just fine with `map`: it accepts any number of lists:

[17]:
```
list(map(lambda x, y: x * y, [1, 2, 3, 4], [5, 1, 5, 1]))
```

[17]:
```
[5, 2, 15, 4]
```

The above won't work with our `mymap`. We'd have to go and learn some extra concepts for it so we won't do that here.

1.0.4 Higher-order functions

Notice that when using `map`, we are passing-in a *function*. Specifically, we are passing-in a function such as `lambda x: x**2`, or the name of a function, such as `sq`. We don't pass in a function *call*, i.e. we don't write `map(sq(10), ...)`.

Any function which treats other functions as "just another data type", e.g. by taking functions as arguments or by returning functions, is called a *higher-order function* (HOF). `map` is the best-known HOF.

`max`, `min`, `L.sort`, `sorted`, and some other functions are also higher-order because as we know we can pass a *key function* to them, e.g.:

[1]:
```
sorted([-10, -5, 0, 5], key=lambda x: x**2)
```

[1]:
```
[0, -5, 5, -10]
```

1.0.5 Callbacks

A *callback* is a function you supply to some other piece of code, in the knowledge that it will be called sometime later, not under your control. We have already seen some callbacks, e.g. when we passed a key function to `sorted` above, we knew that it would be called many times with various arguments.

Callbacks are common in two particular situations:

- In GUI programming, e.g. if we create a button then we also create a function `on_click` which will be called *by the GUI framework* every time the user clicks it. Example: https://blog.kivy.org/2014/03/kivys-bind-method/
- In long-running algorithms, such as optimisation algorithms including neural network training algorithms, we can often customize the output that is printed during the run by passing in a callback. Example: https://keras.io/models/sequential/#fit_generator

1.0.6 Don't Repeat Yourself

The "don't repeat yourself" (DRY) principle is "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" – Hunt and Thomas, *The Pragmatic Programmer*.

For a simple example, what could go wrong here?

```
def customer_id(L, x):
    if x in L:
        return L.index(x)
    else:
        return -1 # default value

# 1000 lines later...

if customer_id(L, x) == -1:
    print("Customer does not exist")
```

It is not obvious, but `-1` is duplicated. The danger here is that someone editing `customer_id()` might decide on a different default value, e.g. `None`, but not update the other use. DRY tells us to eliminate the duplication – represent the default value in only one place – to avoid that danger.

1.0.7 DRY, re-use and functional programming

"if you find yourself copying and pasting a block of code, you have probably found an opportunity for functional abstraction" – http://wla.berkeley.edu/~cs61a/fa11/lectures/functions.html

Functional programming is especially suitable for DRY. The ability to pass one function to another allows us to write very reusable functions so that we don't have to copy and paste.

Overall, in functional programming the aim is again to write very *lego-like* programs: small pieces which can fit together in many different ways. This allows for maximum *re-use* of code.

Further reading (advanced): recall again our code for Newton's method for square roots. Walk through http://wla.berkeley.edu/~cs61a/fa11/lectures/functions.html#example-newton-s-method to see how it can be generalised in a highly DRY way, using functional programming ideas, to find logarithms and other real functions.