Lecture 5.2 : Immutable and mutable function arguments

Introduction

- We look at the options available for passing arguments to functions.
- We look at the implications of passing mutable compared to immutable arguments.
- We look at default and keyword parameters.
- We look at a dangerous trap that you should not fall into.

Immutable arguments

- Below we attempt an alternative approach to returning a value from a function.
- We compute an answer and write it to the the argument passed to the function.
- Will it work? Will the function caller see the answer?
- Let’s try it and see!

```python
1   def celsius2fahrenheit(temperature):
2       temperature = temperature * 1.8 + 32
3
4   temperature = 21
5   celsius2fahrenheit(temperature)
6   print(f'That is {temperature:.1f} degrees Fahrenheit')
```

```
That is 21.0 degrees Fahrenheit
```

- Hmm. That did not work. Why not?
- On line 5 we passed an argument `temperature` to the function.
- This `temperature` is a reference to the number `21`.
- On line 1 a *new* local variable called `temperature` is created (it is local to the `celsius2fahrenheit()` function) and into it is copied the reference contained in the `temperature` created on line 4.
- Thus we effectively have two `temperature` variables. One is global (line 4) and one is local (line 1).
- **Each of these is a reference to the same immutable number**.
- Inside the function on line 2 we *overwrite* the local variable `temperature` with a *new* reference to a *new* number.
- However, this update is *invisible* to the caller of the function and has *no effect* on the contents of the global `temperature` which continues to reference the number 21.
- Note that it is always only a reference that is copied from an argument to a parameter and *no new copy* of the underlying object is created.
- The argument and the parameter are instead *aliases* for the same object.

Mutable arguments

- By contrast, when we pass a *mutable* argument to a function then the function *can* make changes to it that are visible to the caller.
- The crucial difference here is that the called function *does not overwrite the reference* passed to it *but instead writes through the reference* passed to it to update the underlying object.

```
1  def add2list(alist):
2     alist.append(99)
3
4  blist = [1, 2, 3]
5  add2list(blist)
6  print(blist)
```

```
[1, 2, 3, 99]
```

- When we pass the argument `blist` to the `add2list` function the reference in `blist` is copied into the parameter `alist`.
- Thus both `blist` and `alist` reference the same object.
- When we execute line 2 we write *through* the `alist` reference to append to the underlying object.
- On returning to `main` the change is visible as we have written *through* and *not overwritten* the reference passed to the function.

Default parameter values

- It is possible in Python to assign a default value to a function parameter.
- If the function call does not supply a corresponding argument then the parameter is assigned its default value for that invocation of the function.
- If a corresponding argument is supplied then its value overrides the default value for that invocation of the function.
- Thus parameters in the function definition with associated default values are *optional* while parameters in the function definition without associated default values are *required*.
- **Parameters with default values must appear rightmost in the parameter list in the function definition.**

Keyword arguments

- By default, arguments are mapped according to their position to corresponding parameters.
- It is however also possible to map arguments to parameters using keywords.
- Thus it is possible to order arguments differently to parameters as long as `parameter=argument` pairs are supplied in the function call.
- Any “traditional” arguments to the left of any `parameter=argument` pairs are matched by position with parameters.
- **Any parameter=argument pairs must appear rightmost in the argument list in the function call.**

- **Multiple values for a parameter are obviously not allowed. (How would Python know which one to use?!)**

Exercise

- Provide the output of the following code (or indicate an error where applicable).

```
def arithmetic(a, b, c=3, d=4):
   return a + b + c + d

print(arithmetic(1, 2, 5, 6))

print(arithmetic(3, 4, 5))

print(arithmetic(3, 4))

print(arithmetic(3, 4, d=3))

print(arithmetic(b=5, a=4, d=2, c=1))

print(arithmetic(a=2, b=4, 6))

print(arithmetic(6, a=2, b=4))

print(arithmetic(b=2, a=4, c=6))

print(arithmetic(b=5, 2, 5))
```

The mutable default parameter value trap

- Consider the output of the following.

```
def add_animal(animal, zoo=[]):
  zoo.append(animal)
  return zoo

animals = add_animal('leopard')
print(animals)

animals = add_animal('giraffe', ['hyena'])
print(animals)

animals = add_animal('tarantula')
print(animals)
```

```
['leopard']
['hyena', 'giraffe']
['leopard', 'tarantula']
```

- The output above is surprising.
- As programmers we do not like surprises.
- Passing a second argument to `add_animal` is optional.

- If no argument is supplied then the corresponding parameter takes on the value `[]` i.e. the empty list.
- We can see that when we first call the function and pass it `'leopard'` it hands back the list `['leopard']`.
- This makes sense as `zoo` defaults to the empty list.
- The second time we call the function we pass it `'giraffe'` and a list `['hyena']` and it hands us back the list `['hyena', 'giraffe']`.
- So far so good. This all makes sense.
- In the final call we pass the function `'tarantula'` and we expect to be returned the list `['tarantula']` after `zoo` defaults to the empty list.
- Instead we get back the list `['leopard', 'tarantula']`.
- Why is that? What's going on? I'm surprised and I don't like it!
- Clearly the `zoo=[]` default assignment of the empty list, when no corresponding argument is supplied, does not work as intended.
- The problem is the this default value, an empty list **is initialised only once** by Python.
- It is initialised when the function `def` is first encountered.
- This means that the default mutable values *have a memory* and anything added to them will stay there.
- **Never not use mutable types as default parameter values.**
- We fix the problem as follows.

```python
def add_animal(animal, zoo=None):

  # Create a new empty list each time one is required
  if zoo is None:
     zoo = []

  zoo.append(animal)
  return zoo

animals = add_animal('leopard')
print(animals)

animals = add_animal('giraffe', ['hyena'])
print(animals)

animals = add_animal('tarantula')
print(animals)
```

```
['leopard']
['hyena', 'giraffe']
['tarantula']
```

- That's more like it! Now I can sleep easy.