Lecture 5.2 : Function parameters and 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.

Immutable arguments as parameters

- Below we have attempted an alternative approach to returning a value from a function. Will it work? Let's try it and see:

```
import sys

def celsius2fahrenheit(t):
    t = t * 1.8 + 32

def main():
    temp = float(sys.argv[1])
    celsius2fahrenheit(temp)
    print('That is {:.2f} degrees Fahrenheit'.format(temp))

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

```
$ python3 celsius_03.py 30
That is 30.00 degrees Fahrenheit
```

- Hmm. That did not work. Why not? The program's behaviour can be explained as follows. We pass the argument `temp` to the function. The contents of `temp` are *copied* into the parameter `t`. The `temp` variable contains a reference to the *immutable* float 30. Thus after the copy both `temp` and `t` contain a reference to same the *immutable* float 30. The `t` parameter behaves like any local variable. Inside the function we *overwrite* this local variable with a *new* reference to a *new* float. However, this update is *invisible* to the caller of the function and has *no effect* on the contents of `temp`. The `temp` variable continues to reference the float 30.

- 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 as parameters

- What happens when we pass a *mutable* object to a function? Explain the behaviour of the following program:

```
import sys
```

```
def add2list(alist):
    alist.append(99)

def main():
    mylist = [1, 2, 3]
    add2list(mylist)
    print('List contents: {}'.format(mylist))

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

```
$ python3 scope_02.py
List contents: [1, 2, 3, 99]
```

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

- Even with mutable objects, however, there is no guarantee that changes will be visible in the caller. Consider the behaviour of the following program:

```
import sys

def add2list(alist):
    alist = alist + [99]

def main():
    mylist = [1, 2, 3]
    add2list(mylist)
    print('List contents: {}'.format(mylist))

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

```
$ python3 scope_03.py
List contents: [1, 2, 3]
```

- When we pass the argument `mylist` to the function `add2list` the reference in `mylist` is copied into the parameter `alist`. Thus both `mylist` and `alist` reference the same object. When we execute the highlighted line of code we *overwrite* the reference in `alist` to point to a *new* list object. (Contrast this with the previous example where we used `append` to write *through* the supplied reference.) The original object referenced by `mylist` is unaffected. On returning to `main` the change is invisible as the function has simply *overwritten* its reference to the original list (to point to a new list object).

Default values and parameter keywords

- 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.**
- 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=value` pairs are supplied in the function call. This feature can help clarify code where a parameter list is particularly long.
- **Any parameter=value pairs must appear rightmost in the argument list in the function call.**
- Any "traditional" arguments to the left of any `parameter=value` pairs are matched by position.
- Provide the output of the following program (or indicate an error if applicable):

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

def main():
    # a=1, b=2, c=5, d=6, 2-1+5+6 = 12
    print(arithmetic(1, 2, 5, 6))

    # a=3, b=4, c=5, d=4, 4-3+5+4 = 10
    print(arithmetic(3, 4, 5))

    # a=3, b=4, c=3, d=4, 4-3+3+4 = 8
    print(arithmetic(3, 4))

    # a=3, b=4, c=3, d=3, 4-3+3+3 = 7
    print(arithmetic(3, 4, d=3))

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

    # Error: all parameter=value pairs must be rightmost
    print(arithmetic(a=2, b=4, 6))

    # Error: a=6, a=2, b=4, how can a have two values?!
    print(arithmetic(6, a=2, b=4))

    # a=4, b=2, c=6, d=4, 2-4+6+4 = 8
    print(arithmetic(b=2, a=4, c=6))

    # Error: all parameter=value pairs must be rightmost
    print(arithmetic(b=5, 2, 5))

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

```
$ python3 default.py
12
10
8
7
4
Error
Error
8
Error
```

The default mutable value trap

- Provide the output of the following program:

```
def add2list(w, alist=[]):
    alist.append(w)
    return alist

def main():
    word = 'cat'
    nlist = add2list(word)
    print(nlist)
    word = 'pigeon'
    nlist = add2list(word, ['hen'])
    print(nlist)
    word = 'spider'
    nlist = add2list(word)
    print(nlist)

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

```
$ python3 mutablearg.py
['cat']
['hen', 'pigeon']
['cat', 'spider']
```

- The behaviour above is interesting. We see that the second argument to `add2list` 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 `'cat'` it hands back the list `['cat']`. The second time we call the function we pass it `'pigeon'` and a list `['hen']` and it hands us back the list `['hen', 'pigeon']`. So far so good. This all makes sense. In the final call we pass the function `'spider'` and we expect to be returned the list `['spider']` but instead we get back the list `['cat', 'spider']`. Why is that?!

- Clearly the `alist=[]` default assignment of the empty list, when no corresponding argument is supplied, does not work as intended. This empty list *is initialised only once* by Python. It is initialised when the `def` is first encountered. This means that the list *has a memory* and anything added to it will stay there.

- **Do not use mutable data types as default values.**

- We fix the problem as follows:

```
def add2list(w, alist=None):
    if alist is None:
        alist = []
    alist.append(w)
    return alist

def main():
    word = 'cat'
    nlist = add2list(word)
    print(nlist)
    word = 'pigeon'
    nlist = add2list(word, ['hen'])
    print(nlist)
    word = 'spider'
```

```
    nlist = add2list(word)
    print(nlist)

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

```
$ python3 mutableargfix.py
['cat']
['hen', 'pigeon']
['spider']
```

- That’s more like it!