C7 Functions

Functions

1. normal function

```
def main():
    #call the function in main
    kitten()

#a function
def kitten():
    print('Meow.')
```

2. any default args must after the common args which means they need to be at the end of the lists

```
def main():
    kitten(5, 6, 7)

def kitten(a, b = 1, c = 0):
    print('Meow.')
    print(a, b, c)
```

3. mutable object might be changed but it will affect on the caller

```
def main():
    x = 5
    kitten(x)
    print("in main: x is {x}")

def kitten(a):
    print('Meow.')
    print(a)

#Output:
#Meow.
#5
```

4. name : a special variable name which will return the current name of the module

```
#if this file has been imported in another as a module, this name would have the name of the module
if __name__ == '__main__': main()
```

Arguments

1. (*) is the variable length argument list, useful when have different number of arguments —→ no need to define how many args are required

```
def main():
    kitten('meow', 'grrr', 'purr')

def kitten(*args):
    #len: return the number of items in a container.
    if len(args):
        for s in args:
            print(s)
    else: print('Meow.')

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

Kwargs

1. (**) keyword arguments symbols. Can call the value by its keyword

```
def main():
    #dictionary
    x = dict(Buffy = 'meow', Zilla = 'grr', Angel = 'rawr')
    kitten(**x)

def kitten(**kwargs):
    if len(kwargs):
        for k in kwargs:
            print('Kitten {} says {}'.format(k, kwargs[k]))
    else: print('Meow.')

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

Generator

1. generator will return a string of value

```
def main():
    for i in inclusive_range(25):
        print(i, end = ' ')
    print()

def inclusive_range(*args):
    numargs = len(args)
    start = 0
    step = 1

    # initialize parameters
    if numargs < 1:
        raise TypeError(f'expected at least 1 argument, got {numargs}')
    elif numargs == 1:
        stop = args[0]
    elif numargs == 2:
        (start, stop) = args
    elif numargs == 3:
        (start, stop, step) = args
    else: raise TypeError(f'expected at most 3 arguments, got {numargs}')

    # generator
    i = start
    while i <= stop:
        #yield = return but only used in generator
        yield i
        i += step

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

Decorator

1. a special type of function that returns a wrapper function
2. a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure
3. usually called before the definition of a function you want to decorate
4. It is advisable and good practice to always use functools.wraps when defining decorators in order to avoid lost of metadata
5. ensures that your code is DRY(Don't Repeat Yourself)

```python
import time

def elapsed_time(f):
    def wrapper():
        t1 = time.time()
        f()
        t2 = time.time()
        print(f'Elapsed time: {(t2 - t1) * 1000} ms')
    return wrapper


@elapsed_time
def big_sum():
    num_list = []
    for num in (range(0, 10000)):
        num_list.append(num)
    print(f'Big sum: {sum(num_list)}')

def main():
    big_sum() #only call big_sum function in main because elapsed_time already embedded to the big_sum function

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