Exercise 9 – Advanced Collections

Objective

We'll write a generator function and enhance it. This is an extension to the previous "Functions" chapter, as well as exercising our knowledge of generators and Python in general. If we have time, using a custom-built module and a dictionary comprehension.

Questions

1. By now, you should have an acquaintance with the built-in function range(). You will note that it only works on integers. This exercise is to write a version that handles floating-point numbers – float objects.

   * Please call your program gen.py – we will be using this in later exercises! *

   We won't be implementing everything that range() uses, we will make the first two parameters mandatory. We won't return a range object either, we will implement as a generator (a range object is a generator with extra features).

   Implement a version of range()called frange() with the following signature:

   frange(start, stop[, step])

   The default step should be 0.25. Note: pay attention to the possibility of a mischievous user supplying a step of zero.

   Test with the following code:

```
print(list(frange(1.1, 3)))
print(list(frange(1, 3, 0.33)))
print(list(frange(1, 3, 1)))  # Should print [1.0, 2.0]
print(list(frange(3, 1)))     # Should print an empty list
print(list(frange(1, 3, 0)))  # Should print an empty list
print(list(frange(-1, -0.5, 0.1)))

for num in frange(3.142, 12):
```

```
print(f"{num:05.2f}")
```

Finally:

```
print(frange(1,2))
```

should show something like this:

```
<generator object frange at 0x.....>
```

2. Enhance the **frange** function implemented in the previous exercise. The **range** function allows a single argument to be supplied that signifies the end of the sequence, the start then defaults to zero and the step defaults as before. Implement this in your **frange** function.

   Test with something like this:

```
one = list(frange(0, 3.5, 0.25))
two = list(frange(3.5))
if one == two:
    print("Defaults worked!")
else:
    print("Oops!  Defaults did not work")
    print("one:", one)
    print("two:", two)
```

3. This exercise is a further refinement of **frange**. It is the nature of floating-point numbers that inaccuracies appear, and you probably noticed some in your results. The inaccuracies are so serious that, as it stands, the function is not robust enough for a production environment.

   There are several solutions; one is to use the **decimal** module from the standard library. For that, we need to convert our function arguments to objects of the Decimal class but convert the result back to a float when we yield.

   The Decimal class constructor takes an integer or a string – this gives it the required precision. So, we need to convert our input parameters, for example:

```
step = decimal.Decimal(str(step))
```

   Don't forget to import the **decimal** module and to yield a float. You should find the test results a little more sensible.

If time allows...

4. We have a small module written using the Python C API which iterates through the processes running on Windows. You might have to install this module; your instructor will give you guidance if that is the case. There is also a Linux version, but that one is pure Python.

   The advantage of this module is that it returns the parent process identifier (PPID) of each process, whereas conventional tools like **tasklist** do not.

   The module is called **getprocs**, and the interfaces required for this exercise are:

getfirstproc	Start iteration of running processes.
getnextproc	Next iteration of running processes.

   Both functions take no parameters and return a tuple containing three items: (*integer process id*, *integer parent process id*, *string executable name*). False is returned at the end of the iteration.

   Write a Python program which has a generator function called **igetprocs**. It should yield the same tuple as returned by the **getprocs** interfaces. Make sure that the first process is not omitted! You will need to write test code for **igetprocs**, we suggest using a **for** loop.

5. Using **igetprocs** written in the previous exercise, use a dictionary comprehension to generate a dictionary where the keys are the process ids and the values are each a list containing the parent process id and executable name. The purpose of such a dictionary would be to lookup the parent process of any process id (you may extend this exercise to do that if you have time).

   Hints:

   This is simpler than the example shown in the course material - no **if** statement is required.

   If you get the error "ValueError: too many values to unpack" then consider unpacking. That is, prefix the tuple name with an asterisk.

Solutions

Here are our versions of these exercises, remember that yours can be different to these, but still correct. If in doubt, ask your instructor.

1. Here's the first try at the frange() function (note do not use this in production code!):

```
def frange(start, stop, step=0.25):
    curr = float(start)
    while curr < stop:
        yield curr
        curr += step
```

2. This implements the enhancement to accept a single parameter:

```
def frange(start, stop=None, step=0.25):
    if stop is None:
        stop = start
        curr = 0.0
    else:
        curr = float(start)

    while curr < stop:
        yield curr
        curr += step
```

3. This is a more robust version of frange, using the decimal module:

```
import decimal

def frange(start, stop=None, step=0.25):
    step = decimal.Decimal(str(step))

    if stop is None:
        stop = decimal.Decimal(str(start))
        curr = decimal.Decimal(0)
    else:
        stop = decimal.Decimal(str(stop))
        curr = decimal.Decimal(str(start))


    if step != 0:
        while curr < stop:
            yield float(curr)
```

```
            curr += step
```

4. Here's our solution for **igetprocs:**

```
import getprocs

def igetprocs():

    retn = getprocs.getfirstproc()
    yield retn

    while retn:
        retn = getprocs.getnextproc()
        if retn:
            yield retn


for proc in igetprocs():
print(proc)
```

5. The dictionary comprehension is straightforward:

```
pids = {pid:value for pid, *value in igetprocs()}
print(pids)
```