CSCI2202 Lecture 4: Functional Programming

TAs: Ehsan Baratnezhad (ethan.b@dal.ca); Precious Osadebamwen (precious.osadebamwen@dal.ca)

Overview

- Return in functions
- Types of arguments in functions
- List & Dictionary Comprehensions
- Functional Programming (pure functions, side effects)
- Functions as variables (lambda functions, map-reduce)
- Recursion
- Iterators and itertools
- Generator functions

Let’s refresh how we get things out of functions

Refresher: how a function is executed

```python
# demo.py
def func(z):
    y = z + 10
    return y
```

```python
# script.py / notebook
from demo import func
y = 10 + 1
q = func(y)
```

1. Load definition of `func` from `demo` module into global namespace (i.e., memory)
2. Execute script line by line
3. Encounter `func`:
   a. Look up `func` definition (i.e., code)
   b. Create local namespace for `func` and pass in variable `y` from global namespace
   c. Execute func line by line
   d. Stop when no more code **OR** `return` `
4. Continue executing script line by line from where you jump to `func`

No code run in function AFTER it encounters return

```python
def func(list_of_values):
    for value in list_of_values:
        return value + 1
    print(“Will not run usually”)
    # unless list_of_values is empty


y = func([1, 2, 3])
# y == 2
y = func([]) # will print to screen
# y == None
```

```python
def fixed_func(list_of_values):
    new_values = []
    for value in list_of_values:
        new_values.append(value + 1)
    return new_values


y = fixed_func([1, 2, 3])
# y == [2, 3, 4]
```

Functions always return (implicitly or explicitly)

```python
# demo.py
def func(z):
    y = z + 10
    # return
```

```python
# script.py / notebook
from demo import func
y = 10 + 1
y = func(y)
print(y == None)
```

- Function finishes executing:
  - If line with `return` and a value/variable will be passed back to global namespace
  - If line with just `return` - python will return a special value called None (which can be coerced as a boolean False)
  - If code just ends python automatically adds `return` implicitly and returns None

Print is not return

```
def dbl(x):
    return x * 2

def happy(input):
    y  = dbl(input)
    return 2 * y

>>> z = happy(4)
# z == 16
```

```
def trbl(x):
    print(2 * x)
    # return None - this is IMPLICIT

def sad(input):
    y = trbl(input)
    return 2 * y
>>> z = sad(4)
8
TypeError: unsupported operand type(s) for *:
'int' and 'NoneType
```

There are also some fancy ways we can get information into functions

Functions can take positional or keyword args (kwargs)

```
def fun(a, b, c):
    print(a + b * c)
fun(1, 2, 10)
21


fun(2, 1, 10)
12
```

```
def fun2(name=None, age=0):
    print(f"name={name}, age={age}")


fun2(age=7, name='ordering')
name=ordering, age=7
```

- Kwargs can be optional and have default values
- Kwargs can be set to mutable values but this gets confusing/messy fast (e.g., name=x)

Unpacking an iterable (i.e., lists/sets) of positional args

```
def func(a, b, c, d, e):
    return a, b, c, d, e
listvars = [1,2,3,4,5]
func(listvars[0], listvars[1],
    listvars[2], listvars[3],
    listvars[4])
```

```
func(*listvars)
listvars = [1,2]
func(*listvars)
TypeError: func() missing 3 required positional arguments: 'c', 'd', and 'e'
```

Unpacking a dictionary into keyword arguments

```
def fun2(name=None, age=0):
    print(f“name={name}, age={age}”)

vardict = {‘name’: ‘Python’, ‘age’: 36}
fun2(name=vardict[‘name’],
     age=vardict[‘age’])
“name=Python, age=36”
```

```
fun2(**vardict)
name=Python, age=36

vardict[‘extra’] = None
fun2(**vardict)
TypeError: fun2() got an unexpected
keyword argument 'extra'

del vardict[‘extra’]
del vardict[‘name’]
fun2(**vardict)
“name=None, age=36”
```

Warning: avoid mutable default arguments

```
def list_append(e, L=[]):
      L.append(e)
      return L
list_append('x', ['y', 'z'])
  ['y', 'z', 'x']
list_append('a')
  ['a']
list_append('b')
  ['a', 'b']
list_append('c')
  ['a', 'b', 'c']
```

- kwarg is defined as a mutable variable it can be modified!
- As code runs the list stored in L is changed leading to unexpected behaviour
- Can fix this by adding conditional and immutable kwarg definition

```
def list_append(e, L=None):
      if L == None:
            L = []
      L.append(e)
      return L
list_append('x', ['y', 'z'])
 ['y', 'z', 'x']
list_append('a')
 ['a']
list_append('b')
 ['b']
```

Functions can be defined to take variable numbers of args

```
def fun(x, *args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)
    # implicit return None


fun(1, 2, 3, a=4, b=5)
  Positional arguments: (2, 3)
  Keyword arguments: {'a': 4, 'b': 5}
```

- If you include * before a positional argument when **DEFINING** a function it will read the variables in those positions as a tuple of variables
- Similarly ** for kwargs when **DEFINING** the function will read keyword-variables in this position as a dictionary
- You can combine regular arguments with *pos and **kwargs:
  - Style is for regular args to go first, then *pos, and then **kwargs

Python tends to have concise alternatives to write common operations (like modifying elements of a list)

List comprehensions: syntactic convenience

```python
list_x = [1,2,3]

list_y = []

for x in list_x:

    list_y.append(x * 2)

# list_y == [2, 4, 6]
```

```python
list_x = [1,2,3]

list_y = [x * 2 for x in list_x]

# list_y == [2, 4, 6]
```

[**expression** for **variable** in **sequence**]

`x * 2` `x` `list_x`

returns a list, where expression is computed for each element in sequence assigned to variable

List comprehensions: multiple variables and iterables

```
points = [(3, 4), (2, 5), (4, 7)]

multi = [(x, y, x*y) for (x, y) in points]

[(3, 4, 12), (2, 5, 10), (4, 7, 28)]
```

[**expression** for **tuple** in **sequence**]
`(x,y,x*y)` `x,y` `points`

```
[(x, y) for x in range(1, 3) for y in range(4, 6)]

[(1, 4), (1, 5), (2, 4), (2, 5)]
```

[expression for v1 in s1
for v2 in s2
for v3 in s3…]

List comprehensions: conditional filtering

```
[x for x in range(1, 101) if x % 7 == 1
and x % 5 == 2]

[22, 57, 92]



[(x,y,x*y) for x in range(1,11) if 6<=x<=7
for y in range(x,11) if 6<=y<=7 and not
x==y]

[(6, 7, 42)]
```

[expression for v1 in s1 if condition]

- List comprehensions handy but if complicated become hard to read
- **Comprehensions hard to comprehend!**
- If more than simple operation: use explicit loop/functions

Dictionary comprehensions

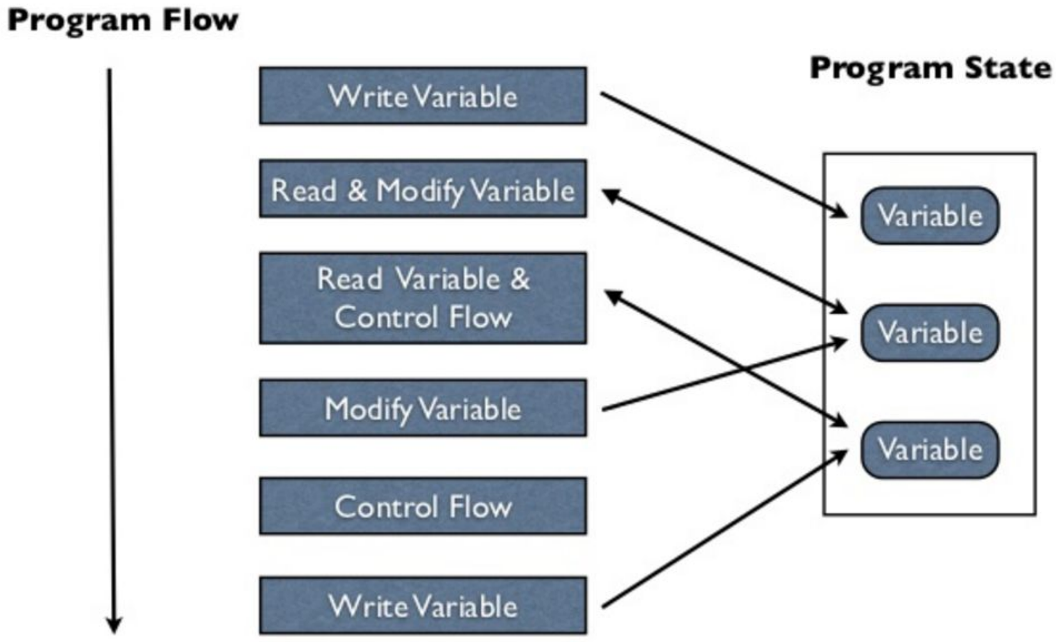
```
names = ['Mickey', 'Donald', 'Scrooge']

list(enumerate(names, start=1))

[(1, 'Mickey'), (2, 'Donald'), (3, 'Scrooge')]

dict(enumerate(names, start=1)) # construct dict from pairs

{1: 'Mickey', 2: 'Donald', 3: 'Scrooge'}

{name: idx for idx, name in enumerate(names, start=1)}

{'Mickey': 1, 'Donald': 2, 'Scrooge': 3}
```

{key : value for variable in list}

- Support conditionals and nesting (identical to list comprehensions)
- Great for basic stuff but again be careful with fancy comprehensions

Functional programming gives us powerful tools for programming with functions

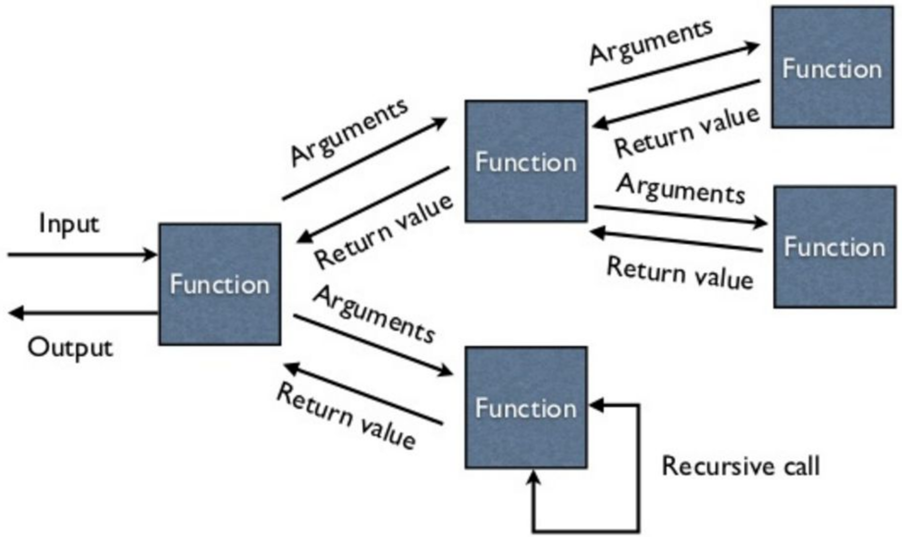
Python is primarily an imperative language



In Imperative languages code is written that specifies a **sequential of instructions** that complete a task. These instructions typically **modifies program state** until the desired result is achieved.

Variables typically represent **memory addresses that are mutable** (can be changed) by default.

Functional programming is built on pure functions



In functional programming individual tasks are small and achieved by passing data to a function which returns a result. This function typically does not change the state of the system or other functions.

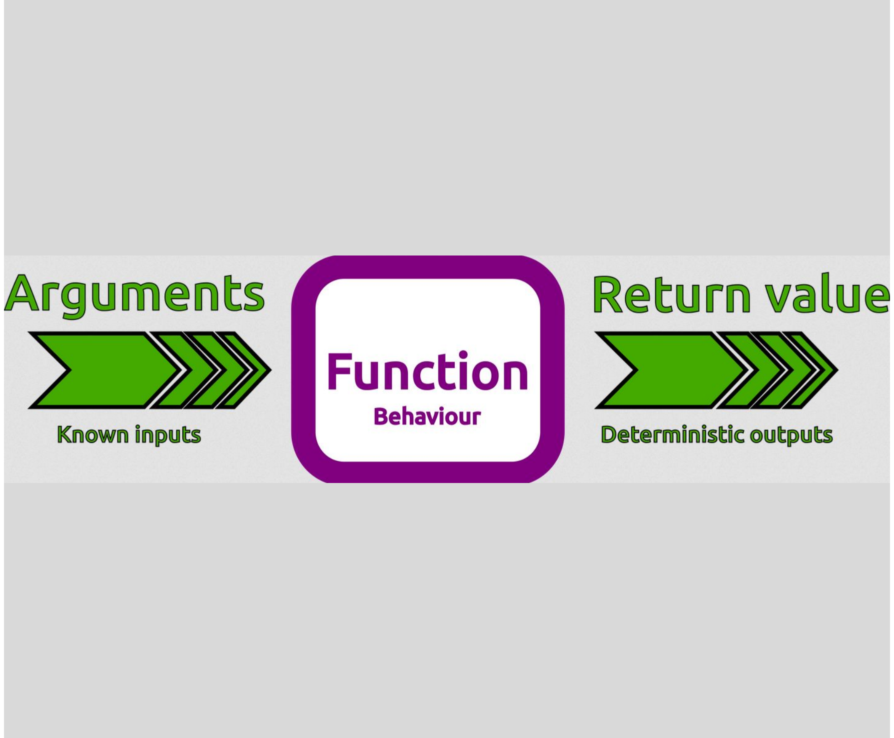
Functions are **composed** together to form more complex tasks. These composed functions pass the result of their evaluation to the next function, until all functions in the composition have been evaluated.

The entire functional program can be thought of as a single function defined in terms of smaller ones.

Program execution is an **evaluation of expressions**, with the nesting structure of function composition determining program flow.

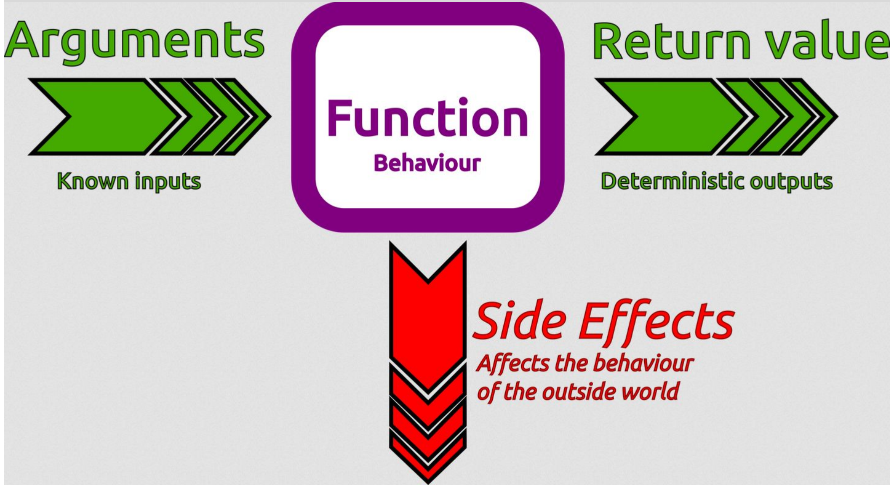
Variables are typically **immutable** and represent values (in the mathematical sense).

Pure functions have defined input and output



- Ideally a function is a simple box with clearly defined interactions with the rest of the script **AND** system:
  - Only way for information to enter the function via arguments
  - Only way for information to leave the function via the return values (i.e., an effect)
- Functions that do this are known as **PURE** functions
- These have more predictable behaviours that let us combine functions in fancy ways

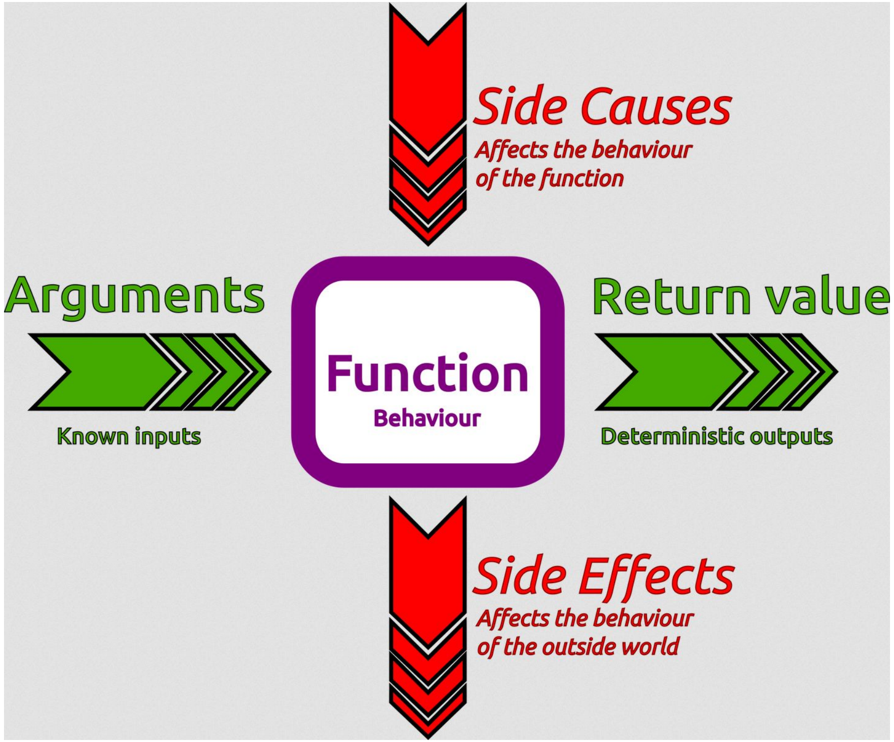
Side effects remove purity but are often useful!



- Side-effects are when a function can change things e.g.,
  - `print` in a function is a side-effect
  - changing a value across namespaces
  - Creating/deleting a file

```
y = [1, 2, 3]
def func(x):
    y[1] = ‘a’
    print(“string”)
    return x
z = func(5) # z == 5
 “string”
y # y == [1, ‘a’, 3]
```

Side causes can make things complicated



- Side-causes are when a function is changed by things other than arguments:
  - `random seed`
  - Computer resource usage
  - Moving/missing files on computer
  - Global variables

```
import random

random.seed(42)

def func():

    return random.randint(0, 100)
```

Python is NOT a functional language but does support some functional approaches.

Single expression function can be defined using `lambda`

```
x = lambda a, b : a * b

print(x(5, 10))

50

func2 = lambda x: len(x) / 2

func2([1,2,3,4])

2.0
```

A lambda function is a small **anonymous** function.

A lambda function can take any number of arguments, but can only have **one expression**.

Lambda functions **RETURNS** whatever the expression evaluates

Example of lambda for custom sorting of a list

```
L = [ 'AHA', 'Oasis', 'ABBA', 'Beatles', 'AC/DC', 'B. B. King', 'Bangles', 'Alan Parsons']

# Sort by length, secondary after input position (default, known as stable)

sorted(L, key=len)

['AHA', 'ABBA', 'Oasis', 'AC/DC', 'Beatles', 'Bangles', 'B. B. King', 'Alan Parsons']

# Sort by length, secondary alphabetically

sorted(L, key=lambda s: (len(s), s))

['AHA', 'ABBA', 'AC/DC', 'Oasis', 'Bangles', 'Beatles', 'B. B. King', 'Alan Parsons']
```

Functions can be treated like any other variable (most of the time)

Functions are just a special type of variable

```
def func_var1(y):
    return y * 10

def func_var2(y):
    return y / 10

def func(func_var, x):
    return func_var(x)
```

```
func(func_var1, 50)
500
func(func_var2, 50)
5
func(lambda x: x / 5, 50)
10
```

You can pass a function as an argument to a function!

map is a function that applies a function to a list of inputs

```
my_pets = ['alfred', 'tabitha',
'william', 'arla']

uppered_pets = []

for pet in my_pets:

     pet_ = pet.upper()

     uppered_pets.append(pet_)

print(uppered_pets)
```

```
# syntax: map(func, *iterables)

# func is the function on which each
element in iterables (as many as they
are) would be applied to



uppered_pets = map(str.upper, my_pets)

# map is lazy

# uppered_pets == map object at xxx

uppered_pets = list(uppered_pets)
```

filter lets us just keep items where a func is True

```
scores = [66, 90, 68, 59,
76, 60, 88, 74, 81, 65]


def is_A_student(score):
    return score > 80


over_80 = filter(is_A_student,
                 scores)
print(list(over_80))
[90, 88, 81]
```

```
dromes = ("demigod", "rewire", "madam",
"freer", "anutforajaroftuna", "kiosk")


x = filter(lambda word: word == word[::-1],
           dromes))


print(x)
“madam”
```

`reduce` lets us cumulatively apply a function

```python
from functools import reduce

numbers = [3, 4, 6, 9, 34, 12]

def custom_sum(a, b):
    return a + b


result = reduce(custom_sum, numbers)

print(result)

68
```

We can also have functions return functions

```
def make_power_func(n):
    return lambda x: x ** n


power_5 = make_power_func(5)
power_5(99)
9509900499
```

Remember whenever you see lambda you can replace it with a full def

Functions calling themselves is particularly powerful

Calculating factorials: one option iteration

```
def factorial(n):
    # initialize result
    result = 1
    # multiply each number between 1 and n
    for current_num in range(1, n+1):
        result = result * current_num
    return result
```

n! = n * (n-1) * (n-2) * … * 1

When we use a loop - this is called "iteration"

We can break down factorials into smaller factorials:

n! = n * (n-1)!

(n-1)! = (n-1) * (n-2)!

0! = 1

Calculating factorials: recursive functions

```python
def factorial(n):
    # base case: n equals zero
    if n == 0:
        return 1
    # recursive case: n > 0
    else:
        return n * factorial(n-1)
```

Recursive function are functions which include themselves as part of its definition.

Need to determine:

- the **recursive case** (i.e., $n! = n * (n-1)!$)
- the **base case** (i.e., $0! = 1$)

Calculating factorials: recursive functions

```
def factorial(n):

    # base case: n equals zero

    if n == 0:

        return 1

    # recursive case: n > 0

    Else:

        return n * factorial(n-1)
```

```
factorial(3):

    return 3 * factorial(2)
```

```
factorial(2):

    return 2 * factorial(1)
```

```
factorial(1):

    return 1 * factorial(0)
```

```
factorial(0):

    return 1
```

Calculating factorials: recursive functions

```
def factorial(n):
    # base case: n equals zero
    if n == 0:
        return 1
    # recursive case: n > 0
    Else:
        return n * factorial(n-1)
```

```
factorial(3):
    return 3 * factorial(2)
```

```
factorial(2):
    return 2 * factorial(1)
```

```
factorial(1):
    return 1 * 1
```

Calculating factorials: recursive functions

```
def factorial(n):
    # base case: n equals zero
    if n == 0:
        return 1
    # recursive case: n > 0
    Else:
        return n * factorial(n-1)
```

```
factorial(3):
    return 3 * factorial(2)
```

```
factorial(2):
    return 2 * 1
```

Calculating factorials: recursive functions

```
def factorial(n):
    # base case: n equals zero
    if n == 0:
        return 1
    # recursive case: n > 0
    Else:
        return n * factorial(n-1)
```

```
factorial(3):
    return 3 * 2
```

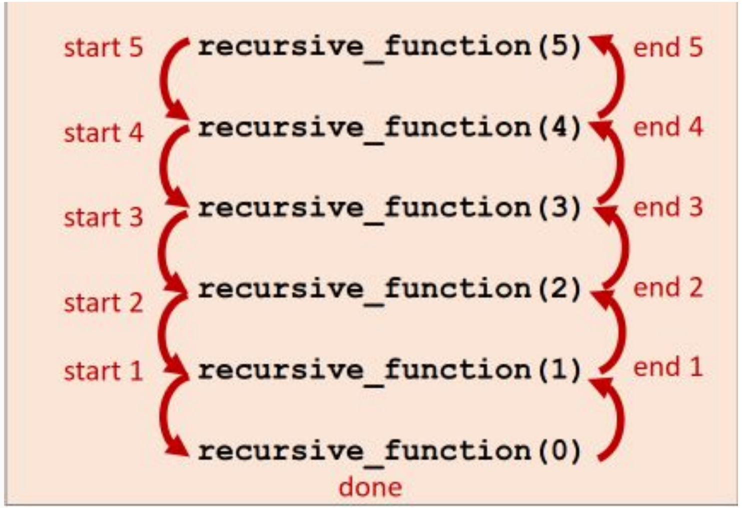
Calculating factorials: recursive functions

```
def factorial(n):
    # base case: n equals zero
    if n == 0:
        return 1
    # recursive case: n > 0
    Else:
        return n * factorial(n-1)
```

```
factorial(3):
    return 6
```

More recursions

```
def recursive_function(x):
    if x > 0:
        print("start", x, end='; ')
        recursive_function(x - 1)
        print("end", x, end='; ')
    else:
        print("done")
recursive_function(5)
start 5; start 4; start 3; start 2; start 1; done
end 1; end 2; end 3; end 4; end 5
```




```
recursive_function(1000000)
RecursionError: maximum recursion
depth exceeded
```

Iterators and itertools

Lists/strings/tuples/dict are all iterators

```
L = ['a', 'b', 'c']
it = iter(L) # calls L.__iter__()
next(it) # calls it.__next__()
 'a'
next(it)
 'b'
next(it)
 'c'
next(it)
 StopIteration
```

- Lists are iterable (must support **__iter__**)
- iter returns an iterator (must support **__next__**)
- `next(iterator_object)` returns the next element from the iterator, by calling the iterator_object.__next__(). If no more elements to report, raises exception `StopIteration`
- `next(iterator_object, default)` returns default when no more elements are available (no exception is raised)
- for-loops, comprehensions, map-reduce require iterable objects

Itertools provides a lot of useful functions for iterators

```
from itertools import combinations

bills = [20, 20, 20, 10, 10, 10, 10,
10, 5, 5, 1, 1, 1, 1, 1]

for combo in combinations(bills, 3):

    print(combo)

 (20, 20, 20)

 (20, 20, 10)

 (20, 20, 10)
```

A choice of ***k*** things from a set of ***n*** things is called a **combination**,

`itertools.combinations()` function takes two arguments:

an **iterable**

a positive integer **n**

returns:

an iterator with tuples of all

combinations of **n** elements in original **iterable**.

Combinations and permutations often useful in science!

```
from itertools import permutations

list(permutations(['a', 'b', 'c']))

 [('a', 'b', 'c'), ('a', 'c', 'b'),

  ('b', 'a', 'c'), ('b', 'c', 'a'),

  ('c', 'a', 'b'), ('c', 'b', 'a')]
```

An ordered group of ***k*** things from a set of ***n*** things is called a **permutation**,

`itertools.permutations()` function takes two arguments:

an **iterable**

a positive integer **n**

returns:

an iterator with tuples of all

permutations of **n** elements in original **iterable**.

Generator functions use yield instead of return

```
def two():
     yield 1
     yield 2
two()
<generator object two at 0x03629510>
t = two()
next(t)
 1
next(t)
 2
next(t)
 StopIteration
```

- A generator function contains one or more yield statements
- Python automatically makes a call to a generator function work as an iterator (for i in t / next(t))
- Calling a generator function returns a generator object
- Whenever next is called on a generator object, the executing of the function continues until the next yield **expr** and the value of **expr** is returned as a result of **next**
- Reaching the end of the function or a return statement, will raise **StopIteration**
- Once consumed, can't be reused

More generator examples

```
def my_generator(n):
    yield 'Start'
    for i in range(n):
        yield chr(ord('A') + i)
    yield 'Done'
```

```
g = my_generator(3)
print(g)
 <generator object my_generator at 0x03E2F6F0>
print([x for x in g])
 ['Start', 'A', 'B', 'C', 'Done']
print(list(g)) # generator object g exhausted
[ ]
```

- Generators are lazy
- Cannot be reused (only if a new generator object is created, starting over again)

More generator examples

```
def my_range(start, end, step):
    x = start
    while x < end:
        yield x
    x += step


list(my_range(1.5, 2.0, 0.1))
 [1.5, 1.6, 1.7000000000000002,
1.8000000000000003, 1.9000000000000004]
```

Summary

- Functions stop executing on return (no return means implicit `return None`)
- Functions can take positional and keyword arguments (and variable lengths)
- Comprehensions are convenient ways of creating iterables
- Functions can be used as variables and in functions (higher-order functions)
- Recursion calling themselves can be a useful way of breaking down problems
- Iterable variables are anything you can iterate over (itertools provide useful tools for these variables)
- Generator functions use `yield` and will lazily create a series of outputs as they are iterated over