Computer Programming I

Lecture 2: Functions and Modules

Hans Karlsson

Last time

- Expressions
- Variables
- Basic operators
- Conditional statements (if/else/elif)
- Loops (while/for)

Today’s lecture

- Functions and procedures
  - Definition of functions
  - Calling functions
  - Terminology
  - Parameters and arguments
  - Parameter passing
  - Return values, multiple return values with tuples
  - Functions with functions as argument for flexible and powerful solutions
- Modules
  - Importing modules
  - Definition of modules
  - Modules as abstraction
- Generators

Functions/procedures in Python

A function is defined/introduced by:

```
def function_name(parameter1, parameter2, …):
    function body

def fn(a, b):
    return a * b
```

Single parameter functions

```
def square(x):
    return x * x

y = square(2)
print(y)
>> 4
y = square(2.5)
print(y)
>> 6.25
```

Single parameter functions

```
def factorial(n):
    assert(type(n) is int)
    res = 1
    for i in range(1, n+1):
        res = res * i
    return res

y = fac(5)
print(y)
>> 120
```

Single parameter functions

```
def factorial(n):
    assert(type(n) is int)
    res = 1
    for i in range(1, n+1):
        res = res * i
    return res

y = fac(5)
print(y)
>> 120
```

```
y = square(5.0)
print(y)
>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fac
AssertionError
```

Multiple parameter functions

```
a = 2.0
b = 3.0

def mystery(a, b):
    if b < a:
        return b
    else:
        return a

print(mystery(b, a))
>> ???
```

What happens when the code runs?

1) '3.0' is printed
2) '2.0’ is printed
3) 'None' is printed
4) Error

Multiple parameter functions

```
a = 2.0
b = 3.0

def minimum(a, b):
    if b < a:
        return b
    else:
        return a

print(minimum(b, a))
>> 2.0
```

What happens when the code runs?

1) '3.0' is printed
2) '2.0' is printed
3) 'None' is printed
4) Error

Parameter passing - example 1

```
y = 1

def f(x):
    x = x + 1
    return x

z = f(y)

print(f'{y}, {z}')
>> ???
```

What happens when the code runs?

1) '1, 2'
2) '2, 2'
3) '2, 1'
4) Error

Parameter passing - example 1

```
y = 1

def f(x):
    x = x + 1
    return x

z = f(y)

print(f'{y}, {z}')
>> 1, 2
```

What happens when the code runs?

1) '1, 2'
2) '2, 2'
3) '2, 1'
4) Error

Parameter passing - example 2

```
x = 1

def f(x):
    x = x + 1
    return x

y = f(x)

print(f'{x}, {y}')
>> ???
```

What happens when the code runs?

1) '1, 2'
2) '2, 2'
3) '2, 1'
4) Error

Parameter passing - example 2

```
x = 1

def f(x):
    x = x + 1
    return x

y = f(x)

print(f'{x}, {y}')
>> 1, 2
```

What happens when the code runs?

1) '1, 2'
2) '2, 2'
3) '2, 1'
4) Error

Parameter passing - example 3

```
def swap_first_two(list):
    list[0], list[1] = list[1], list[0]
    return list


x = [1, 2, 3, 4, 5]


print(x)
>> [1, 2, 3, 4, 5]
y = swap_first_two(x)
print(x)
>> ??
```

What happens when the code runs?

1) '[1, 2, 3, 4, 5]'
2) '[2, 1, 3, 4, 5]'
3) 'None'
4) Error

Parameter passing - example 3

```
def swap_first_two(list):
    list[0], list[1] = list[1], list[0]
    return list


x = [1, 2, 3, 4, 5]


print(x)
>> [1, 2, 3, 4, 5]
y = swap_first_two(x)
print(x)
>> ??
```

What happens when the code runs?

1) '[1, 2, 3, 4, 5]'
2) '[2, 1, 3, 4, 5]'
3) 'None'
4) Error

Parameter passing by name

Consider the function:

```
def divide(denominator, numerator):
    return numerator / denominator
```

It is possible to call divide as follows:

```
print(divide(denominator=2.0, numerator=1.0))
>> 0.5
```

Parameter passing with default values

Sometimes there may be default values, which we want to use most of the time when we call a function, while providing the possibility to provide a different value when necessary. Then we can define a function:

```
def divide(denominator, numerator = 1.0):
    return numerator / denominator

print(divide(4.0))
>> 0.25
print(divide(4.0, 2.0))
>> 0.5
```

Ducktyping! If it walks like a duck...

Parameter types in Python

Just as variables in Python have no type, parameters are also typeless.

Functions can be written once and for all, but called with parameters of different data-types. We saw that in the first example square, which only requires that multiplication is defined for the data-type of the argument.

```
def square(x):
    return x * x
```

This enables you to write very general functions that work with any data-type satisfying the required assumptions.

Return values

In Python, with its dynamic typing, it is fully possible to write code like this:

```
def f(x):
    if x > 0.5:
        return 2 * x
    else:
        return f'{x}'
```

but it is poor style to do so. Whoever calls y = f(x), must start by checking what type the return value y has before they can do any processing with y, which they may forget to do.

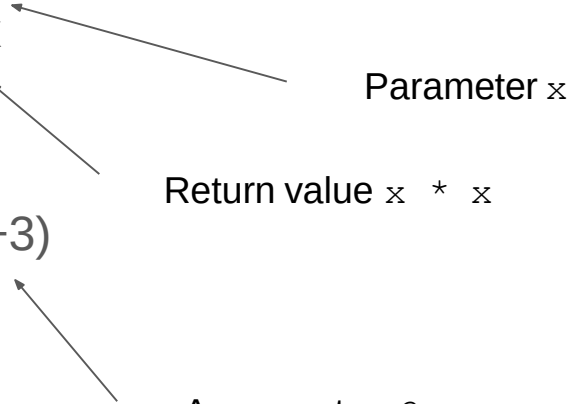
Functions without return values (procedures)

```
def print_balance(name, balance):
    print(f'{name} has {balance} in their bank account.')

print_balance('Johan', 512)
>> Johan has 512 in their bank account.
r = print_balance('Johan', 512)
>> Johan has 512 in their bank account.
print(type(r))
>> <class 'NoneType'>
print(r)
>> None
```

All functions return something:
None is returned if no explicit return statement is reached before the functions reaches the end.

Terminology

```
def square(x):
    return x * x

y = 1
z = square(y+3)
```



Parameter - variables that receive the passed argument.

Argument - the value being passed at an actual call of the function.

Functions with functions as parameters/arguments

```
def bubble_sort(lst, cmp = less):
    n = len(lst)-1
    for n in range(n, 0, -1):
        for i in range(n):
            if cmp(lst[i+1], lst[i]):
                lst[i+1], lst[i] = lst[i], lst[i+1]


xs = [19, 1, 4, 8, 2, 9, 5, 13, 7]
```

Functions with functions as parameters/arguments

```
def less(x, y):
    return x < y

def greater(x, y):
    return y < x

print(type(less))
>> <class 'function'>
```

Functions with functions as parameters/arguments

```
def bubble_sort(lst, cmp = less):
    n = len(lst)-1
    for n in range(n, 0, -1):
        for i in range(n):
            if cmp(lst[i+1], lst[i]):
                lst[i+1], lst[i] = lst[i], lst[i+1]
```

```
xs = [19, 1, 4, 8, 2, 9, 5, 13, 7]
print(xs)
>> [19, 1, 4, 8, 2, 9, 5, 13, 7]
bubble_sort(xs, less)
print(xs)
>> ??
```

Bubble sort debugging

```
def bubble_sort_debug(lst, cmp = less):
    n = len(lst)-1
    print(lst)
    for n in range(n, 0, -1):
        for i in range(n):
            if cmp(lst[i+1], lst[i]):
                lst[i+1], lst[i] = lst[i], lst[i+1]
        print(lst)
```

[3, 2, 8, 1, 5, 9, 0]
After 1 iterations:
[2, 3, 1, 5, 8, 0, 9]
After 2 iterations:
[2, 1, 3, 5, 0, 8, 9]
After 3 iterations:
[1, 2, 3, 0, 5, 8, 9]
[1, 2, 0, 3, 5, 8, 9]
[1, 0, 2, 3, 5, 8, 9]
[0, 1, 2, 3, 5, 8, 9]

Functions with functions as parameters/arguments

```
def bubble_sort(lst, cmp = less):
    n = len(lst)-1
    for n in range(n, 0, -1):
        for i in range(n):
            if cmp(lst[i+1], lst[i]):
                lst[i+1], lst[i] = lst[i], lst[i+1]
```

```
xs = [19, 1, 4, 8, 2, 9, 5, 13, 7]
print(xs)
>> [19, 1, 4, 8, 2, 9, 5, 13, 7]
bubble_sort(xs, less)
print(xs)
>> [1, 2, 4, 5, 7, 8, 9, 13, 19]
```

Functions with functions as parameters/arguments

```
def bubble_sort(lst, cmp = less):
    n = len(lst)-1
    for n in range(n, 0, -1):
        for i in range(n):
            if cmp(lst[i+1], lst[i]):
                lst[i+1], lst[i] = lst[i], lst[i+1]
```

```
xs = [19, 1, 4, 8, 2, 9, 5, 13, 7]
print(xs)
>> [19, 1, 4, 8, 2, 9, 5, 13, 7]
bubble_sort(xs, greater)
print(xs)
>> [19, 13, 9, 8, 7, 5, 4, 2, 1]
```

Functions with multiple return values

In Python, functions can easily return multiple value by returning a tuple:

```
def quad_equation(p, q):
    discriminant = p*p - 4*q
    if discriminant >= 0:
        d = math.sqrt(discriminant)
        x1 = (-p + d)/2.0
        x2 = (-p - d)/2.0
        return x1, x2
    else:
        return None  #
```

Functions with multiple return values

In Python, functions can easily return multiple value by returning a tuple:

```
roots = quad_equation(-3, 2)
print(roots)
>> (1.0, 2.0)
r1, r2 = quad_equation(-3, 2)
print(r1, r2)
>> 1.0 2.0
```

Return exits the function

`return` exits the function (and assigns the return value).

```
def abs_value_with_print(x):
    if(x >= 0.0):
        return x
    print(f'{x}')
    return -x
```

```
print(abs_value_with_print(2.0))
>> 2.0
```

```
print(abs_value_with_print(-2.0))
>> -2.0
2.0
```

Errors (that are not handled) exits the function

```
def divide(denominator, numerator):
    if denominator == 0:
        raise ValueError('Zero denominator in division.. Booo')
    return numerator / denominator

divide(0, 2.0)
>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Zero denominator, boo
```

Functions inside functions

```
def smooth(list, alpha):
    def comp_y(y, x):
        return y * alpha + (1.0 - alpha) * x
    y_val = 0.0
    for i in range(len(list)):
        y_val = comp_y(y_val, list[i])
        list[i] = y_val
```

Inner functions can be useful if you want to define and give a name to an operation that is used within the outer function.

Methods - functions associated with objects

Methods are functions associated with objects (unlike free-standing functions). You may have seen these:

```
t1.forward(30)
```

forward moves the turtle that the variable t1 refers to 30 pixels ahead.

The object before the dot is the object that the method is called on.

```
t2.forward(30)
```

forward moves the turtle that the variable t1 refers to (which from this code may be the same turtle object or a different object) If the method is not defined for the data-type, we get an error message (e.g. “AttributeError: 'str' object has no attribute 'forward'”).

Modules

A module is a Python file (.py) containing a collection of definitions (functions and classes, etc) and variables. A module can be run as a script or imported into another module/script.

By calling:

python my_program.py

my_program.py will be called as a script.

Modules with functions

As an example, consider a module (my_math.py) with a single function

```
def factorial(n):
    res = 1
    for i in range(1, n+1):
        res = res * i
    return res
```

Importing definitions

We may now access the function factorial from my_math.py in another module (my_script.py) in a few ways:

1)

```
from my_math import factorial
print(factorial(7))
>> 5040
```

2)

```
import my_math as mm
print(mm.factorial(7))
>> 5040
```

Importing a module is running a script

If you write code outside of functions in a module, and import this module, the code will run just as if you executed the script from the terminal.

test_module.py:

```
x = 5
print(x)
```

Importing test_module.py from another module:

```
import test_module as tm
>> 5
```

Importing a module is running a script

This is why you rarely want to include code outside of functions inside module, in case someone wants to or needs to import the module into another module/script without being forced to run that code. One solution if we want a module to both work as a module and a script:

```
def main():
    print(5)

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

Modules as abstractions

One of the primary ways of creating reusable code packages is to collect functions and classes in **modules**.

By gathering code related to a certain problem or domain in a module, you can structure a program code to be managable, discoverable, and maintainable.

Since functions in a module need to import functions from other modules, for them to be visible, you may not want to split closely related functions that call each other in too many different modules.

A real-life example of modules - ex: py_alpha_amd

A real python library for solving image registration problems in image analysis was structured in the following set of modules:

distances (alpha_amd.py, symmetric_amd_distance.py, ...)
filters (filt.py)
generators (mask_generators.py, noise_generators.py)
optimizers (gd_optimizer.py, adam_optimizer.py)
transforms (affine_transform.py, rigid_transform.py, rotate_2d_transform.py, ...)

Modules example: py_alpha_amd

transforms (affine_transform.py, rigid_transform.py, rotate_2d_transform.py, ...) contain various geometric transformations. They are naturally grouped in a module.

The transformations are very similar in structure and some have a dependency on others which make them belong together both code-wise, and also conceptually.

Generators - example with prime number generation

```
def primality_test(x):
    for i in range(2, x):
        # if x is divisible by i, x is not prime
        if x % i == 0:
            return False
    return True
def prime_list(n):
    list = []
    for x in range(2, n+1):
        if primality_test(x):
            list.append(x)
    return list
```

Comment!

Generators - example with prime number generation

```
def primality_test(x):
    for i in range(2, x):
        # if x is divisible by i, x is not prime
        if x % i == 0:
            return False
    return True

def prime_generator(n):
    for x in range(2, n+1):
        if primality_test(x):
            yield x
```

yield freezes the execution and returns x. When the next value is requested, the function continues on the next line until another value can be produced or until the function returns.

Generators - example with prime number generation

```
for p in prime_generator(1000):
    print(p, end=' ')
```

```
>> 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193
197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307
311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421
431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547
557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797
809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929
937 941 947 953 967 971 977 983 991 997
```

Generators - example with prime number generation

```
# create a generator
pgen = prime_generator(100)
print(next(pgen))
print(next(pgen))
print(next(pgen))
print('Hello')
print(next(pgen))
print(next(pgen))
```

```
>> 2
3
5
Hello
7
11
```

Infinite lists with generators

```
def prime_generator_infinite():
    x = 2
    while True:
        if primality_test(x):
            yield x
        x = x + 1
```

Infinite loop. If this was a function, it would never return.

Because the generator freezes its execution at yield, we can define an infinite sequence of prime numbers without enumerating them all. Each prime number is produced only on demand.

Infinite lists with generators

```
pgen_inf = prime_generator_infinite()
for i in range(1000):
    print(next(pgen_inf), end=' ')
```

```
>> 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109
113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239
241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379
383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521
523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661
673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827
829 839 ......
```

Summary

Functions are a fundamental building block in Python used to encapsulate units of logic, mathematics, or procedural operation as a named entity which can be used and reused with both different values (arguments) but also for different data-types, reducing the amount of code duplication needed.

Modules are groups of functions and other definitions gathered in a file (or group of files) exposing a conceptual unit of functionality.

Generators allow to write elegant code for representing sequences (possibly infinite) which can be directly used in for example for-loops.