L2a-thiqu264-lensc874

October 11, 2019

1 Laboration 2A

2 Introduction

In this first part of the lab, we will be exploring * Functions * How functions are called. * Argument passing * Return values. * Function usage * Construction of simple multi-function programs. * Functions that work on several kinds of inputs (ie simple polymorphism via duck typing).

Additionally we will touch upon * Exceptions and * simple assertion testing and debugging.

This lab might require you to search for information on your own to a larger extent than in lab 1. As in the last lab, Lutz' Learning Python and the official documentation might be helpful. Also make sure to make use of the available lab assistance!

3 Handin instruction

This lab consists of two parts, 2A and 2B (on functional and declarative patterns). After you've finished lab 2A, hand it in straight away (using the name 2a as the lab name). You do not need to wait until you've finished lab 2B.

3.1 Functions in Python

a) Write a function that takes a radius and returns area of a circle with that radius. What would be a good name for the function and the argument? Python has a value for π in a certain standard library module. Which might that be? Don't type in the constant yourself.

```
In [1]: import math
        def circle_area(radius):
            return math.pi * radius**2
```

[Hint: Google. Or consider modules we have `imported` previously.]

b) How would you call the function, if you wanted to calculate the area of a circle with radius 10cm?

```
In [23]: "Area of circle in square centimeters: " + str(circle_area(10))

Out[23]: 'Area of circle in square centimeters: 314.1592653589793'
```

c) How would you call the function using named arguments/keyword arguments?

```
In [21]: "Area of circle in square centimeters: " + str(circle_area(radius = 10))

Out[21]: 'Area of circle in square centimeters: 314.1592653589793'
```

[Note: In this case, the calling of the function is somewhat artificial. When writing scripts or working with programs that take several parameters, this style can be quite useful. This sidesteps questions of if this particular library takes the input or the output as the first argument, or the like. The code of course becomes more verbose.]

d) Write a function `circle_area_safe(radius)` which uses an if statement to check that the radius is positive and prints `The radius must be positive` to the screen if it is not, and otherwise calls the `circle_area` function. Also, if the radius is not positive the `circle_area_safe` function should signal to the code calling it that it has failed by returning `None`.

```
In [5]: def circle_area_safe(radius):
            if radius <= 0:
                print("The radius must be positive")
                return None
            else:
                return circle_area(radius)
```

e) Recreate the `circle_area_safe` function (call this version `circle_area_safer`) but instead of printing a message to the screen and returning `None` if the radius is negative, *raise* a ValueError exception with suitable error message as argument.

```
In [11]: def circle_area_safer(radius):
             if radius > 0:
                 return(circle_area(radius))
             else:
                 raise ValueError("The radius should be positive.")
```

f) To test out how functions are called in Python, create a function `print_num_args` that prints the number of arguments it has been called with. The count should not include keyword arguments.

```
In [12]: def print_num_args(*args):
             print(len(args))
         print_num_args(1,2,3)  # Should print the number 3.

3
```

g) Write a function `print_kwargs` that prints all the keyword arguments.

```
In [14]: # Your definition goes here
         def print_kwargs(*args, **kwargs):
             print("The 2 regular arguments are:")
             for i, arg in enumerate(args):
```

```
            print(str(i) + ": " + str(arg))
        print()
        print("And the keyword arguments are (the ordering here is arbitrary):")
        for key, value in kwargs.items():
            print(str(key) + " is set to ", str(value))

print_kwargs("alonzo", "zeno", foo=1+1, bar = 99)

# source: http://book.pythontips.com/en/latest/args_and_kwargs.html
```

```
The 2 regular arguments are:
0: alonzo
1: zeno

And the keyword arguments are (the ordering here is arbitrary):
foo is set to  2
bar is set to  99
```

h) Below we have a very simple program. Run the first cell. It will succeed. What happens when you run the second cell, and why? In particular, consider the error produced. What does it mean. What value has been returned from the function, and how would you modify the function in order for it to work?

```
In [15]: def my_polynomial(x):
             print(x**2 + 30 + 225)

         polyval = my_polynomial(100)
```

```
10255
```

```
In [16]: double_the_polyval = 2*my_polynomial(100)
```

```
10255
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-16-b3cd279745e9> in <module>
----> 1 double_the_polyval = 2*my_polynomial(100)


TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

```
In [17]: """The reason an error occures is that the provided function returns an object of typ
         The calculated value will only be printed but not returned. To succeed, a return stat
         be included."""

         def my_polynomial(x):
             return(x**2 + 30 + 225) # Instead of printing, the value will be returned.

         polyval = my_polynomial(100)
         2*my_polynomial(100) # As a result, the function does not return an error.

Out[17]: 20510
```

3.2 Script/program construction (a tiny example)

Regardless of which programming language we use, we will likely construct programs or scripts that consist of several functions that work in concert. Below we will create a very simple Monte Carlo simulation as a basis for breaking down a larger (though small) problem into sensible, (re)usable discrete pieces. The resulting program will likely utilise control structures that you have read about before.

a) The following is a well-known procedure for approximating π: pick n uniformly randomly selected coordinates in an $2R \times 2R$ square. Count the number of the points that fall within the circle of radius R with its center at (R, R). The fraction of these points to the total number of points is used to approximate π (exactly how is for you to figure out). (Note that this is not to be confused with MCMC.)

Write a program consisting of several (aptly selected and named) functions, that present the user with the following simple text user interface. The yellow text is an example of user input (the user is prompted, and enters the value). It then prints the results of the simulations:

`pi_simulation()`

Welcome to the Monty Carlo PI program!

Please enter a number of points (or the letter "q" to quit): 100 Using 100 points we (this time) got the following value for pi: 3.08 This would mean that tau (2xPI) would be: 6.16

Please enter a number of points (or the letter "q" to quit): 100 Using 100 points we (this time) got the following value for pi: 3.12 This would mean that tau (2xPI) would be: 6.24

Please enter a number of points (or the letter "q" to quit): q

Thank you for choosing Monty Carlo.

[**Note**: This is a task largely about program structure. Unless there are substantial performance drawbacks, prefer readability over optimisation.]

An important part of programming is to allow a reader who is perhaps unfamiliar with the code to be able to understand it, and convince themselves that it is correct with respect to specification. There should also be as few surprises as possible.

```
In [18]: import math
         import random

         def generate_point(radius):
             x = random.uniform(0, 2*radius)
             y = random.uniform(0, 2*radius)
```

```
        return (x, y)

    def point_in_circle(radius, point):
        # Getting center of the circle.
        center = (radius, radius)
        # Using the Pythagorean theorem, the distance of each point to the center is calc
        leg1 = abs(point[0] - center[0])
        leg2 = abs(point[1] - center[1])
        hypotenuse = math.sqrt(leg1**2 + leg2**2)
        # If hypotenuse <= radius, then point lies in circle.
        if hypotenuse <= radius:
            return 1
        else:
            return 0

    def pi_simulation(radius = 1):
        # Printing greeting.
        print("Welcome to the Monty Carlo PI program!")
        # Performing program until letter q is inserted from user.
        n_points = 0
        while n_points != "q":
            # Defining number of simulations.
            print()
            n_points = input('Please enter a number of points (or the letter "q" to quit)
            # Stopping function if n_points is defined as q.
            if n_points ==  "q":
                print()
                print("Thank you for choosing Monty Carlo.")
                return
            # Initializing number of simulated points in circle.
            points_in_circle = 0
            # Simulating.
            for sim in range(int(n_points)):
                point = generate_point(radius)
                points_in_circle += point_in_circle(radius, point)
            # Calculating pi.
            pi = points_in_circle * 4 / int(n_points)
            # Printing result.
            print("Using " + n_points + " points we (this time) got the following value fo
            print("This would mean that tau (2xPI) would be: " + str(2 * pi))

    pi_simulation()
```

```
Welcome to the Monty Carlo PI program!

Please enter a number of points (or the letter "q" to quit): 10
Using 10 points we (this time) got the following value for pi: 1.6
This would mean that tau (2xPI) would be: 3.2
```

```
Please enter a number of points (or the letter "q" to quit): 100
Using 100 points we (this time) got the following value for pi: 2.92
This would mean that tau (2xPI) would be: 5.84

Please enter a number of points (or the letter "q" to quit): 10000
Using 10000 points we (this time) got the following value for pi: 3.144
This would mean that tau (2xPI) would be: 6.288

Please enter a number of points (or the letter "q" to quit): q

Thank you for choosing Monty Carlo.
```

[Hint: You might want to consider the function `input`. Try it out and see what type of value it returns.]

b) One feature of Python's simplicity is the possibility to (comparatively) quickly produce code to try out our intuitions. Let's say we want to compare how well our approximation performs, as compared to some gold standard for pi (here: the version in the standard library). Run 100 simulations. How large is the maximum relative error (using the definition above) in this particular run of simulations, if each simulation has $n = 10^4$ points? Is it larger or smaller than 5%? Write code that returns this maximum relative error.

```
In [19]: import math

         # Defining number of simulations.
         n_points = 10**4

         # Initializing list for collection of relative errors
         relative_errors = []

         # Simulating 100 times.
         for sim in range(100):
             # Initializing number of simulated points in circle.
             points_in_circle = 0
             # Simulating.
             for i in range(n_points):
                 point = generate_point(1)
                 points_in_circle += point_in_circle(1, point)
             # Calculating pi.
             pi = points_in_circle * 4 / int(n_points)
             # Appending relative error to list.
             relative_errors.append(abs(pi-math.pi)/math.pi)

         print("Maximum relative error: " + str(max(relative_errors)))
         if max(relative_errors) < 0.05:
             print("Maximum relative error is smaller than 5%.")
```

```
        else:
            print("Maximum relative error is larger than 5%.")
```

```
Maximum relative error: 0.013244029700242443
Maximum relative error is smaller than 5%.
```

[Note: This is only to show a quick way of testing out your code in a readable fashion. You might want to try to write it in a pythonic way. But in terms of performance, it is very likely that the true bottleneck will still be the approximation function itself.]

3.3 Fault/bugspotting and tests in a very simple setting

It is inevitable that we will make mistakes when programming. An important skill is not only to be able to write code in the first place, but also to be able to figure where one would start looking for faults. This also involves being able to make the expectations we have on the program more explicit, and at the very least construct some sets of automatic "sanity checks" for the program. The latter will likely not be something done for every piece of code you write, but it is highly useful for code that might be reused or is hard to understand (due either to programming reasons, or because the underlying mathemetics is dense). When rewriting or optimising code, having such tests are also highly useful to provide hints that the changes haven't broken the code.

Task: The following program is supposed to return the sum of the squares of numbers $0, ..., n$.

```
In [25]: # Do not modify this code! You'll fix it later.

         def update_result(result, i):
             result = result + i*i
             return result

         def sum_squares(n):
             result = 0
             for i in range(n):
                 result = update_result(n, result)
```

a) What mistakes have the programmer made when trying to solve the problem? Name the mistakes in coding or thinking about the issue that you notice (regardless of if they affect the end result). In particular, write down what is wrong (not just "line X should read ..."; fixing the code comes later). Feel free to make a copy of the code (pressing `b` in a notebook creates a new cell below) and try it out, add relevant print statements, assertions or anything else that might help. Note down how you spotted the faults.

```
In [21]: """The function update_result seems to be well-implemented.
         It takes a current result and adds the square of i to this result before returning th

         The function sum_squares instead includes one mistake.
         The input of the function is fine - it just uses n as the input and loops over each e
         The result will be initialized by 0 which is also fine.
         The problem within the coded function is the specified input for the update_result -
         The first parameter should be the current resultand the second argument should be the
```

```
        which the squared value will be added to the result.
        Instead, the wrong input was chosen (n instead of result and result instead of i)
        This will not lead to an error, but to a false result.
        Programmed like this, in every iteration i the value of 0 will be added to the value
        Furthermore, the result is not returned by this function which leads to a return of N
        """
        type(sum_squares(5))
```

```
Out[21]: NoneType
```

b) Write a few simple assertions that should pass if the code was correct. Don't forget to include the *why* of the test, preferably in the error message provided in the `AssertionError` if the test fails.

```
In [26]: def test_sum_squares():

             # Format: ( input, expected output, error message )
             # You may replace these with namedtuples if you want to.

             tests = (
                 (3, 5, "input of 3 should return 5."),
                 (4, 14, "input of 4 should return 14.")
             )

             for arg, output, error_msg in tests:
                 assert sum_squares(arg) == output, error_msg

             print("--- test_sum_squares finished successfully")

         test_sum_squares()
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)

<ipython-input-26-87347263c0bd> in <module>
     14     print("--- test_sum_squares finished successfully")
     15
---> 16 test_sum_squares()


<ipython-input-26-87347263c0bd> in test_sum_squares()
     10
     11     for arg, output, error_msg in tests:
---> 12         assert sum_squares(arg) == output, error_msg
     13
     14     print("--- test_sum_squares finished successfully")
```

```
AssertionError: input of 3 should return 5.
```

[Note: Might there be any corner/edge cases here?]

c) Write a correct version of the code, which conforms to the specification.

```
In [27]: def sum_squares(n):
             result = 0
             for i in range(n):
                 result = update_result(result, i)
             return(result)

         test_sum_squares()   # It should pass all the tests!
```

```
--- test_sum_squares finished successfully
```

[Note: This is also a rather primitive testing strategy, but it is sometimes enough. If we wanted to provide more advanced testing facilities, we might eg use a proper unit test framework, or use tools to do property based testing. This, as well as formal verification, is outside the scope of this course.

Those interested in testing might want to consult the web page for the IDA course TDDD04 Software testing or the somewhat abbreviation-heavy book by Ammann & Offutt, which apparently also features video lectures.]

3.4 Polymorphic behaviour (via duck typing)

In Python we often write functions that can handle several different types of data. A common pattern is writing code which is expected to work with several types of collections of data, for instance. This expectation is however in the mind of the programmer (at least without type annotations), and not something that the interpreter will enforce until runtime. This provides a lot of flexibility, but also requires us to understand what our code means for the different kinds of input. Below we try this out, and in particular return to previously known control structures.

a) Write a function `last_idx` that takes two arguments `seq` and `elem` and returns the index of the last occurrence of the element `elem` in the iterable `seq`. If the sequence doesn't contain the element, return -1. (You may not use built-ins like .find() here.)

```
In [28]: def last_idx(seq, elem):
             last = -1
             for i, val in enumerate(seq):
                 if val == elem:
                     last = i
             return last
```

b) What does your function require of the input? In particular, your answer should include if it would work with a string, a list or a dictionary. In the latter case, what would `elem` be matched against?

```
In [30]: def check_input(input_type, seq, elem):
             if last_idx(seq, elem):
                 print("Works with " + input_type)

         check_input("string","hallo", "l")
         check_input("list",[1,2,1], 1)
         check_input("dictionary", {"First": 2, "Second": 3, "Third": 2}, 2)

         """The function is very flexible related to the input.
         The seq-argument can be of the type of all mentioned types (string, list, dictionary)
         However, in the last case (dictionary), an index will be returned.
         Since within a dictionary the values are not connected to indices but to keys,
         this result actually does not make that much sense."""
```

```
Works with string
Works with list
Works with dictionary
```

```
Out[30]: 'The function is very flexible related to the input. \nThe seq-argument can be of the
```

c) Add some assert-style tests that your code should satisfy. For each test, provide a description of what it tests, and why. That can be made as part of the assert statement itself.

```
In [31]: def test_last_idx():

             # Format: ( (seq, elem), expected output, error message )
             # You may replace these with namedtuples if you want to.

             tests = (
                 ( ([1,2,3,2], 2), 3, "last_idx should return last index, for sequences with se
                 ( ("Assertion","s"), 2, "last_idxt should return the last index for a letter i
                 ( ({"Train1": 1, "Train2": 2, "Train3": 3}, "Train3"), 2, "last_idx should ret
             )

             for args, output, error_msg in tests:
                 assert last_idx(*args) == output, error_msg     # last_idx takes two argument

             print("--- test_last_idx finished successfully")

         test_last_idx()
```

```
--- test_last_idx finished successfully
```

The fact that a program doesn't crash when given a certain input doesn't necessarily ensure that the results are what we expect. Thus we need to get a feel for how eg iteration over different types of data behaves, in order to understand how our function behaves.

d) Can we use `last_idx` with a text file? What would the program try to match `elem` against? What would the return value signify (eg number of words from the start of the file, lines from the start of the file, bytes read...)?

```
In [32]: with open("students.txt", "r") as data:
             for each in data:
                 print(last_idx(each, "1"))

         """The function can be used with a text file. Using the text file "students.txt" from
         for each row the last index with the specified element will be returned.
         Every string element will be matched seperately against elem which means that also fo
         increasing index of 1.
         All in all, the returned value indicates the number of characters from the start of t
         To treat the whole data in once is not possible. It can be only analysed line by line

49
20
-1
49
47


Out[32]: 'The function can be used with a text file. Using the text file "students.txt" from la
```

[Hint: Try it out! Open a file like in lab 1, using a `with` statement, and pass the file handle to the function. What is the easiest way for you to check what the function is comparing?]

3.4.1 Attribution

Lab created by Anders Märak Leffler (2019), using some material by Johan Falkenjack. Feel free to reuse the material, but do so with attribution. License CC-BY-SA 4.0.