SNI SURGICAL NEUROLOGY INTERNATIONAL

OPEN ACCESS

For entire Editorial Board visit : http://www.surgicalneurologyint.com

Editor: **James I. Ausman**, MD, PhD University of California, Los Angeles, CA, USA

Editorial

Programming for physicians: A free online course

Pieter L. Kubben

Department of Neurosurgery, Maastricht University Medical Center, Maastricht, The Netherlands

E-mail: *Pieter L. Kubben - pieter@kubben.nl
*Corresponding author

Received: 15 February 16 Accepted: 15 February 16 Published: 29 March 16

Abstract

This article is an introduction for clinical readers into programming and computational thinking using the programming language Python. Exercises can be done completely online without any need for installation of software. Participants will be taught the fundamentals of programming, which are necessarily independent of the sort of application (stand-alone, web, mobile, engineering, and statistical/machine learning) that is to be developed afterward.

Key Words: Big data, data science, information technology, programming, statistics

Access this article online

Website: www.surgicalneurologyint.com

DOI: 10.4103/2152-7806.179382

Quick Response Code:

INTRODUCTION

Welcome to this crash course in programming, aimed at physicians. Why physicians? Because historically it is not the profession who is most prone to learn computer programming whereas nowadays being able to communicate in computer language (at least to some extent) is a welcome addition to scientific research skills in general, and data science in particular. Earlier I wrote an editorial "Why physicians might want to learn computer programming" which details how I think about this.[3]

In short, computer programming consists of three main components: Data input, data manipulation, and data output. Note that this is a high-level and very general overview. Input can be some text entered by the user, a mouse-click, a tap or gesture on a mobile device, a file loaded from disk, a database from the web, or a sensor in a device. Output can also be some text, graphic, audio, video, or whatever you can imagine that is being sent back to the user or some server.

As you can guess, data manipulation is the interesting part where the magic happens. Or actually, there is not so much magic. Again, there are, roughly speaking, three main components: Calculations, conditional statements, and loops.

Variables are parameters that you can use to store information. When I say "store," I do not implicitly mean "store on disk" or "store online." In the first place, it means to reserve (or as it is called officially, to allocate) some memory on your computer where you can put some information temporarily. Once the code is executed, the variables can be erased from memory as you do not longer need them. If you want to store something for later use, e.g., on disk, USB stick, or remote server, you have to do that explicitly. The latter part is not covered in this crash course, but it is good to know. For now, variables are the sort of parameters you will mostly use for data manipulation.

Calculations are rather self-explanatory. Historically computers were developed as advanced calculators, and only later other functionality was added. Calculations

How to cite this article: Kubben PL. Programming for physicians: A free online course. Surg Neurol Int 2016;7:29.
http://surgicalneurologyint.com/Programming-for-physicians:-A-free-online-course/

can differ from simple math to much more complicated processes, but the essence remains the same.

Conditional statements are exactly what the name says. To a nurse you might say that "if" a patient has a temperature higher than 39°C, then you like to order a blood culture, "else" they should measure temperature again in 2h. Conditional statements or "if-else statements" are useful for decision-making in programs.

Loops are used for repetitive statements. Computers are much better than humans in repeating the same task repeatedly, and loops are the way to do that. There are two sorts of loops, a so-called for-loop and a while-loop. We will cover them in more detail.

As soon as your program grows in size, you will need a way to organize your code. Functions are a way to avoid repeating code, and in bigger projects you can use modules or classes for that purpose.

Note: Modules and classes are not the same thing. While a module is essentially just a file with some functions, classes are using in a programming style called object-oriented programming (OOP). OOP is not a part of this crash course.

For this crash course, we will use the programming language Python.[5] It has a fairly straightforward and readable syntax, is open source and completely free to use, and there are many scientific modules that you can find interesting to use once you mastered the basics.

Getting started

To recap the welcome section, programming consists of data input, data manipulation, and data output. The interesting part is the manipulation, which in itself consist of calculations, conditional statements, and loops (or repetitive statements). In this crash course, input and output are deliberatly very simple (just plain text or numbers). Hence, we can focus on the essence, the manipulation. Let's get started!

At ProgrammingForPhysicians.com (or ProgrammingForDoctors.com if you find that easier to type), there are interactive code editors windows available between the text to experiment directly with some code. You simply type your code and press "Run." At that moment, commands which are typed in the programming language are translated into commands that the computer can actually process. Because as you might know, the computer just understands (so-called binary) code which consists of 0's and 1's. Technically speaking, the code gets compiled. The output is visible in the right side of the window.

I once read a quote on the web saying that programming is like writing a book. Except, "if you forget one comma at page 128 the whole thing makes no sense"… There are strict rules for what is allowed and what not. If you violate one of these rules, the compiler will give you an error message.

Any line that is preceded with a hashtag (#) is ignored by the Python compiler. You can use this to provide (optional) comments in your code that will help you (and your colleagues if you are collaborating) to understand what the code is doing and what you were thinking, if you return to that code at a later moment.

```
# The command below writes some text on the screen
# Press Run to see the result
print 'Hello Doctor'
```

It is a tradition to do a "Hello World" application as your first programming exercise to please the ancient programming gods Kodos and Debuggos. Regardless of the programming language you choose to learn, most educational resources start with this. All it does is print "Hello World" to the screen. I chose a slightly modified approach, as we also have to keep Asklepios pleased. Obviously, you would have guessed that print is the command to send output to the screen. Technically, print is a function, and a function can take one or more arguments. In this case, the argument that is passed to the function consists of some text. The value of the argument is Hello Doctor. If the output is text, you put that between quotes. It does not matter whether you use single quotes as in print 'some text' or double quotes as in print "some text," but you cannot combine them: Print "this goes wrong" will produce an error!

```
# Try to run this code, it will produce an error
print 'This goes wrong!!"
```

On the other hand, this will work well:

```
# Take a look at these valid combinations
# of quote signs
# And Run them, don't just look!;-)
print "Hey, let's play a little with quote signs"
print 'It also works "the other way around" as you can
see'
```

All right, you are ready to move to the next level…:-)

VARIABLES

Single value

Variables are little containers in which you can store some data, bring them to another place, and do something with them. Before you can use a variable you need to create it. Technically, you need to declare the variable and assign a value to it. You do this in Python by typing the variable name, followed by = and then the variable value.

```
# Some variables are being declared here
first_name = 'Pieter'
last_name = "Kubben"
year_of_birth = 1978
is_neurosurgeon = True
```

You can notice a few things from the examples. A variable name consists of text and numbers, but also underscores (_) which I add for readability. Variable names are not allowed to contain spaces, so **first name = 'Pieter'** would produce an error. Also, do not start a variable name with a number! And last but not the least: Variable names are case-sensitive, which means that **First_name** is not the same as **first_name**. The latter applies for functions too, by the way: **print** does exist, **Print** does not.

An alternative solution that is used in many languages is the so-called "camel case" notation: **myFirstName = 'Pieter'**. Have a guess why it is called camel case! (Yes, the animal…).

The choice between using lowercase with underscores or camel case notation is personal, in a sense that the compiler does not care. However, in all programming languages there are conventions, which mean that most people using that language share the same preference. I choose to follow the conventions used in Python and use the underscores.

Important: Just because you can give any name to a variable does not mean you should! is really helpful for code readability to use variable names that make sense!! So **first_name** is better than **fn** because the latter is not clear at all. Yes, you can provide code comments, but your code will quickly become an unreadable mess if you do not have the discipline to use variable names that make sense!

Text values

Just as with the print examples we saw before, text is enclosed between single or double quotes. Choose whatever you want. Talking about text, the official name of this sort of variable is a string. I did not make up the name, so do not blame me for it!

Now, if **print 'Pieter'** is a valid command, and **first_name='Pieter'** is a valid command, then **print first_name** should be a valid command too! And it is!

Numbers

In contrast to a string, a number is not enclosed between quotes. Technically speaking, there are many sorts of numbers, but to keep it simple, you need to know two of them: An integer is a whole number, a floating-point number (or simply: Float) is a number that contains decimals.

```
# Here are some number examples
my_integer = 1
my_float = 1.5
```

As you see, it does not make a difference when declaring the variables. It can make a difference for calculations in the future, and for that reason it is good to know the difference exists.

Now what if you would do **my_number = '1'** instead of **my_number = 1**? In that case, **my_number** would be treated as text instead of a number. Does it make a difference? Yes, it does! Think about it, I will explain it below.

True/false

True/false values are officially called Boolean values, and they can be only true (yes, 1) or false (no, 0). You could as well create a string variable and set the text value to "true" or "false" but using a Boolean value has some advantages which helps to simplify your code and reduce the chance for errors. We will use Boolean values in conditional statements (or if…else statements).

Single value?

Just to make sure there is no misunderstanding, with "single value" I mean that variables that contain a string, number or Boolean value can only contain one single value at a time. As you saw, it is absolutely possible to change the value of a variable to another value, and in Python you can even change between types (although I would not recommend it). For example,

```
# Changing variable type

# String variable
my_variable = 'hello'
print my_variable

# Now we'll make it a number
my_variable = 100
print str(my_variable)
```

And something new… again. That's why I call this a crash course…;-) If you want to print something to the screen, it needs to be a string. Numbers are perfectly fitted for calculations, but you need to convert them to a string if you want to display them. You do that with the built-in function **str()**. Here, **str** is the function name, and the parentheses **()** contain the arguments that are passed to the function. In this case, that is the variable that you want to convert to a string. Now, if the variable is the only output, it will still work. The variable gets converted implicitly. As soon as we are going to combine string and number output, we need to perform this conversion explicitly.

You may be wondering now why we are writing **print 'some text'** instead of **print('some text')** because I said that print is a function too. And you are right! For this crash course, you are working with Python 2.7 and this is a sort of inconsistency within the language. From Python 3.0 and forward this is solved. Don't worry about it.

Working with variables

All right, so far the explanation. Let's practice a little…

```
# We declare some variables and print it
# type your name instead of the dots
```

```
doctor_name = '...'
print 'Hello' + doctor_name
```

What happens here? After the print statement two strings are glued together. Technically this is called string concatenation and the + sign is the concatenation sign. It is on purpose that I add a space after Hello… try to remove it and run the code again, see what happens!

```
# Now we concatenate two strings and place
# a space between them

location = 'optical'
structure = 'nerve'

print location + ' ' + structure
```

You see? Now, I did not tell you the most important aspect about variables until now… they are called variables for a reason! Their value can be changed throughout the program, just as you can change content from any other (real-world) container. Let's elaborate on the previous example!

```
# We declare a variable, print it and change its
# value afterwards. Then print again.

# First round
location = 'optical'
structure = 'nerve'
print location + ' ' + structure

# Second round
location = 'temporal'
structure = 'artery'
print location + ' ' + structure
```

Got it? Now let's try the same with some numbers.

```
# Adding up some numbers

# First round
my_number = 1
result = my_number + my_number
print 'First result: ' + str(result)

# Second round, with number as string
my_number = '1'
result = my_number + my_number
print 'Second result: ' + result
```

Hey, what's happening here? In the first round, the result is 2. When dealing with numbers, the + sign is the mathematical +, so it results in addition. In the second round, the result is 11 and this is correct! Because here we are not adding up numbers, but we are concatenating strings. Just as print 'hello ' + 'doctor' results in "hello doctor", so does print '1' + '1' result in "11".

Cleaning up

But wait a minute… didn't I tell you in the introduction that every variable takes up some space in memory? Yes, it does! So should we clean up after we are done using them? No, not in most languages. Python, as well as many other modern programming languages, uses a technology called automatic garbage collection to release allocated memory space once it is not needed anymore. So you do not need to clean up!

Multiple values

Now, there are also variables that can contain multiple values in one variable. The most important one that we'll use is called a list, but for a little more advanced applications dictionaries are very useful too.

Lists

```
# We will declare a list variable and
# add some values to it

famous_neurosurgeons = ['Cushing', 'Penfield',
'Leksell', 'Yasargil']
print famous_neurosurgeons
```

As you can see, the list famous_neurosurgeons consists of 4 string values, each one enclosed in strings. Regarding the list as a whole, it starts with [, ends with] and all values are separated by a comma. It does not matter what sort of variable you put in a list: Strings, numbers, and Booleans are all okay. You can even put a list as one of the variables in another list. And if you want, you can mix them, so bullshit_list = ['some text', 99, [1, 2, 'more text']] is a valid list, but useless in my opinion. In practice, you'll stick to one sort of variable in one list like in the famous_neurosurgeons example.

Now, why would you use a list instead of separate variables? For several reasons, you can easily add or remove values from the list, and you can easily access every single value in the list (this will be explained in the Loops section). Just as you use a bag to store multiple items, you use a list to store multiple values. It is a lot more convenient and you can handle more data easily!

Now, the items in the list are ordered and for that reason a list can also be called a sequence. All list values have an index and as most programming languages start counting at 0 instead of at 1, the index 0 refers to the 1st item in the list. In general, the index n refers to the $n + 1^{th}$ item in the list. The index number is added behind the list name between brackets, as in famous_neurosurgeons[0]. Now, you can also select a subset of values by this syntax: some_list[3:6]: This selects all values from some_list starting at index 3 (included) until index 6 (excluded). Hence, this will give you the values with index 3, 4, and 5.

```
# Manipulating a list

# Do some selections
famous_neurosurgeons = ['Cushing', 'Penfield',
'Leksell', 'Yasargil']
print famous_neurosurgeons[0] # prints "Cushing"
print famous_neurosurgeons[1:3] # prints "Penfield"
and "Leksell"

# Add an item to the end of the list
```

```
famous_neurosurgeons.append('Spetzler')
print famous_neurosurgeons

# Insert an item at a specified index
famous_neurosurgeons.insert (3, 'Verbiest')
print famous_neurosurgeons

# Remove the last person from the list
famous_neurosurgeons.pop()
print famous_neurosurgeons

# Remove the second person from the list
# (so with index 1 !!)
famous_neurosurgeons.pop (1)
print famous_neurosurgeons

# Get the length of the current list
print 'We now have ' + str (len (famous_
neurosurgeons)) + ' famous neurosurgeons in our list.'

# Sort the list alphabetically
print sorted(famous_neurosurgeons)
```

In this example, you still may choose to work with different variables for each single person. However, if you are going to load a database with 10,000 records, you definitely do not want to do that! In that case, lists or any other sequence will come in as a very useful manner to handle such a sequence of data.

You may have seen that the last command, sorted() did not seem to do anything. Actually it did, but the results were already in alphabetical order. I just included it so you know you can sort lists easily. By default, it is done alphabetically for string values and numerically in increasing order for numerical values.

Dictionaries

Dictionaries are a sort of list with key-value pairs. As you saw, the list-indexes are just numbers. Sometimes it can be useful to have something else. Suppose you want to have a list with the amount of surgical procedures you performed, then my_procedures = [300, 100, 800] does not make any sense. With a dictionary, it could look like my_procedures = {'lumbar spine': 300, 'craniotomies': 100, 'carpal tunnel release': 800} which makes more sense.

```
# A dictionary example

my_procedures = {'lumbar spine': 300, 'craniotomies':
100, 'carpal tunnel release': 800}
print my_procedures
```

I will not elaborate on dictionaries here, but just wanted to let you know that they exist. In case you were wondering if numbers were the only way to index a sequence: No, they are not. Quod erat demonstrandum!

For this crash course, I will leave it at this… you have seen a detailed use of single-value variables, and got some hands-on experience with lists. Now let us move forward and actually do some data manipulation!

DATA MANIPULATION

Calculations

Now, you've read the introductory texts and know how to deal with variables, let's get started doing what we are here for: Making the computer work for us by giving programmatic instructions!

Basic math

```
# Basic math operations

print 5 + 7      #  addition
print 12 - 2     #  subtraction
print 7 * 6      #  multiplication
print 12 / 3     #  integer division (results in
                    float here)
print 10 / 3     #  floating-point division
print 12 ** 2    # squared value
print 7 % 3      #  modulus (remainder after
                    division)
```

I guess that addition, subtraction and multiplication, and squared value are self-explanatory. Regarding division there can be a difference between integer division and floating-point division. In the code windows on this site everything goes as expected, but in some stand-alone Python 2.x versions, it is possible that print 10 / 3 results in 3 instead of 3.333. The reason is that both 10 and 3 are integer values, and some older compilers do not automatically force the result to be a float if this is necessary. There is a good reason for that (integers consume less memory space) but it is not convenient here. You can force a floating-point division manually by typing print 12 / 3.0 (or even print 12 / 3.). In this case, the denominator gets automatically converted ("casted") to a float. It does not matter which part of the equation you cast, if one of the numbers is a float, the result will be a float. And then you can be sure things go as expected.

Modulus is an interesting operation. One of the most frequent uses is to check if a number is even or odd (some_even_number % 2 = 0).

In case you did not notice, we just wrote print 5 + 7 instead of print str(5 + 7). Python is smart enough to do this for you if these numbers are all you want to print. However, if you like to combine this with text, you need to convert the number manually: print 'Result: ' + str(12 + 7).

Now, we did not spend so much time on variables to ignore them afterward. In practice, you will do the calculations with variables in order to reuse the functionality of your program.

```
# Basic math with variables

# Calculate Body Mass Index
body_mass = 80        # in kg
body_length = 1.80    # in m
```

```
body_mass_index=body_mass/ (body_length ** 2)

# Return the result
print body_mass_index
```

I used parentheses around body_length **2 to ensure that the calculations are executed in the order that I intend. Although without parentheses you should get the same result here, it does not hurt to use them in my opinion. I find the code more readable and less error prone this way.

Until now, we have been hard-coding the variables to get the result, but we could also ask them to the user of the program. Now we're talking!

```
# Ask user for input
user_mass = raw_input('What is your body mass in kilogram?')
user_length = raw_input('What is your body length in centimeters?')

# Convert user input (string!!) to integers
body_mass = int(user_mass)
body_length = int(user_length)

# Convert body_length to meters
body_length = body_length / 100

# Calculate Body Mass Index
body_mass_index = body_mass/(body_length ** 2)

# Display result
print 'Body Mass Index: ' + str(body_mass_index)

# That are quite some digits.
# Let's round the result to one decimal
print 'Rounded Body Mass Index: %. 1f' % body_mass_
index
```

How cool is that! Now you can run this piece of code as much as you want and all the work is done automatically… that is the reason we are doing this!

You saw two new things in this example. First, we needed to convert user input (which comes as a string, because the user types text - even if that text is a number) to an integer using the function int(). You could also use float() to convert to a floating-point number, the result here would be the same. Go ahead, try it!

Second, the last line contains some difficult looking code which is actually quite simple. The piece %.1f says that this piece of code should be replaced by the content behind the next % and it should be formatted in 1 decimal (that is what .1 stands for) and the value is of a floating-point type (that is what f stands for). And in that case, the conversion to string is all-inclusive, no need for a separate str(). Now the result is more readable, isn't it?

In the example above, I made very little steps on purpose. You can be more concise by doing this:

```
# A more concise BMI calculator

# Get input
body_mass = int(raw_input('What is your body mass in kilogram?'))
body_length = int(raw_input('What is your body length in centimeters?'))/ 100

# Return BMI
print 'Your (rounded) Body Mass Index is: %. 1f' % (body_mass/(body_length ** 2))
```

The result is exactly the same. Is less really more? It depends… it does not hurt to be concise as long as you do not compromise on readability. If your code becomes hardly readable, chances are that you will introduce errors. And typing that extra line of code will definitely cost less time than debugging your program later on! So the style you choose is personal, but do yourself a favor: Keep it readable! That little amount of extra memory space by adding one or two more variables does practically nothing on the speed of your program. They don't hurt. Crashes and errors do!

If you are combining functions, be careful to pay attention to the number (and position) of parentheses. When I was typing the concise example, I first typed body_mass = int(raw_input('What is your body mass in kilogram?') and got an error. Can you see why? Hint: Compare it with the (working) example code above.

Advanced math

Computers would be not that interesting if they could only do some simple math. Sure, they do it fast and correct, but still…. There is much more to Python (and other programming languages) than just the basic stuff. Consider the whole language as a big cupboard with one main area and many drawers. So far we just looked in the main area, but for more advanced functionality we are going to open one of the drawers. In Python language, such a drawer is called a module and to open it, we need to import it. Then you can take something from the drawer, e.g., a cup. In our case, the cup is called pi and the way we should name it in order for the compiler to understand it, is math.pi. This is called dot notation which is used frequently in programming. So in the analogy, you first do import drawer and then use the cup by typing drawer.cup.

```
# Import math module
import math

# Ask for tumor diameter and calculate tumor volume
# Assume a spherical tumor for this example
tumor_diameter = float(raw_input('Enter tumor diameter in cm:'))
tumor_volume = (4/3) * math.pi * (tumor_diameter /2) ** 3

# Return tumor volume
print 'Tumor diameter: ' + str (tumor_diameter)
print 'Tumor volume: %. 2f cc' % tumor_volume
```

Sure, you could simply use the value 3.14 instead of math.pi, but I just wanted to demonstrate how to access

additional resources. Although **math.pi** is much more accurate, it does not make a difference here because the input value has a much lower accuracy. By the way, I converted **tumor_diameter** to **float** instead of **int** in order to accept decimal values too. When dealing with tumor volumes, I thought it would be important.;-)

If…else statements

As you can work with variables and perform calculations by now, let's make things even more interesting! I already gave an example in the introduction of regular decision-making, which essentially consists of a structure like this: "If" some condition is valid/not valid, then something should happen, "else" some other thing should (maybe) happen. In programming you can use the same structure for decision-making. Here is a minimal example:

```
if some_condition:
    # perform some action
```

There are two things to be noted here. One is the colon (:) at the end of the first line. This is mandatory in a conditional statement. The other is that the second line is indented in relation to the first line. For this indentation, you mostly use the TAB key on your keyboard, which in turn translates this into a (consistent) number of spaces. The code windows in this lesson use two, some other editors use 4. Both are fine. The important thing is that as soon as you have some longer blocks of code there may be an hierarchy in the code, and indentation is the Python way to organize the code. In the example above, **# perform some action** belongs to the **if some_condition** part, and it should only be executed if that condition applies. Hence, it is a sort of sub-level below this condition.

Now, if we are going to add an **else** statement, we need to get one level up, because the else part is of equal value as the if part. You will see that if you type the code, the editor automatically indents the second line. This is triggered by you using the colon at the end of the first line. If you press ENTER, the indentation is kept, which is helpful in general. Using BACKSPACE you can go back to the top level. Here is the structure of an if…else block:

```
if some_condition:
    # perform some action
else:
    # perform some other action
```

When I introduced Boolean values, I promised to get back to them in this section. Time to keep that promise! Take a look at this example and run it!

```
# Conditional statement using Boolean value

high_fever = True

if high_fever:
    print 'Take blood cultures please'
```

This simply checks whether the condition high_fever is present or not. Note that the two possible values, true and false need to start with an uppercase! Change the value for **high_fever** to false and run the example again. Now nothing happens, because there is no else block defined. By the way, if you want to check specifically is a condition is not present, you can do it like this:

```
# Test for absence of a condition

need_to_worry = False

if not need_to_worry:
    print 'Relax, there is no need to worry!'
```

Boolean values are useful but limited to dichotomizing data. Here are your options for more flexible conditions:

- **==**: Equal to (that's a double = sign)
- **!=**: Not equal to
- **>**: Greater than
- **>=**: Greater than or equal to
- **<**: Lesser than
- **<=**: Lesser than or equal to

So here is the example from the introduction in working code. Go ahead and run it a few times with different input values. I convert the input to a float for the same reason as I did in the tumor volume example in the previous section.

```
# Temperature management

# Initialize variables (in celsius)
current_temp = float(raw_input('What is the current temperature (in Celsius)?'))

cutoff_temp = 39

# Create conditional statement
if current_temp >= cutoff_temp:
print 'Take blood cultures please.'
else:
    print 'Measure temperature again in two hours please.'
```

In case you have more options than just two, you can add **elif** statements (which stands for *else if*) between the **if** and the **else** block. You can use as many as you want:

```
# Example with multiple options

# Initialize variables (in celsius)
temp = float(raw_input('What is the current temperature (in Celsius)?'))

# Create conditional statement
if temp >=39:
    print 'Take blood cultures please.'
elif temp >=38:
    print 'Measure temperature again in two hours please.'
elif temp < 36:
    print 'The patient may be too cold.'
else:
```

```
        print 'The temperature is just fine.'

print 'This line gets printed in all four cases.'
```

The order of the statements is important. First, this code checks if temp is larger than or equal to 39. If it is, it performs the action that belongs to this statement and then stops. So if you enter 39.5 as a temperature, it won't mention you to measure the temperature again in 2 h! Suppose you want that, you can combine if-statements and indentation to created nested statements:

```
# We are combining two statements now

# Initialize variables (in Celsius)
temp = float(raw_input('What is the current temperature (in Celsius)?'))

# Create decision logic
if temp >=38:
        if temp >=39:
                print 'Take blood cultures please.'
print 'Measure the temperature again in two hours please.'
else:
        print 'There is no fever present.'
```

What you see here, is that if the temperature is larger than 38, two things happen. First, there is an extra check whether the temperature is also larger than 39, in which case blood cultures are ordered. Second, regardless of whether the temperature is larger than 39, there is the request to measure again.

Nested statements are very useful, but as with many things in life: Don't overdo it! And think whether nesting is the best solution for a problem.

Multiple parameters

So far we only looked at one parameter at a time, although there could be multiple options (conditions) present. You can also combine multiple parameters using and/or to check if both conditions are present, at least one of them, or none.

```
# Looking at two parameters simultaneously

# Example values
high_pulse = True
low_blood_pressure = False

# Now check for both conditions
if high_pulse and low_blood_pressure:
print 'Be careful, the patient may be in shock!'
else:
        print 'Just be careful.'
```

In case you are combining, e.g., 4 statements and the order of combining those matters, you can use parentheses to force a specific order:

```
# Demo for order

if (condition1 or condition2) and (condition3 or condition4):
        # do something useful
```

Got it so far? Great! Because actually this is it! A few subtle things have been left out, but you really mastered conditional statements now! To finish, let's go back to the body mass index (BMI) example from the previous section and combine calculations and conditional statement to create something useful. Well, sort of, at least.

```
# Calculate BMI and give advice

# Get input
body_mass = int(raw_input('What is your body mass in kilogram?'))
body_length = int(raw_input('What is your body length in centimeters?'))/100

# Calculate BMI
bmi = body_mass/(body_length ** 2)
print 'Your body mass index is: %. 1f' % bmi

# Give advice dependent on BMI
if bmi > 26:
print 'You may want to loose some weight.'
elif bmi < 19:
        print 'You may want to gain some weight.'
else:
        print 'You are on a healthy weight.'
```

This tiny application is already a working example of clinial decision support, isn't it? You can imagine that it is not that difficult to create more complex examples using the same structure. That is what I have been doing in NeuroMind![2]

Go ahead and proceed to the next section on loops! (don't worry, no rollercoasters present!)

Loops

You are almost there! Since you've mastered working with variables, performing calculations and creating conditional statements, you can do a lot. However, with loops, you can also easily repeat all those things. There are two sorts of loops: For-loops and while-loops.

For loops

Most of the time you will be using a for-loop, and because an example says more than a lot of words, let's create a piece of code that prints the values 1 up to 100.

```
# A simple for-loop

for i in range(100):
        print i + 1
```

The built-in function range() creates a sequence of numbers. By default, range (*n*) starts at 0 and continues until the value *n* (so, *n* is excluded). Optionally, you can provide a starting value, and a step size. For example:

```
# Go from 50 to 100 with steps of 5
print 'Going up...'
for i in range(50, 100, 5):
        print i
```

```
# Countdown from 20 to 0 with steps of -2
print 'And going down.'
for i in range(20, 0, -2):
    print i
```

The variable name i is a habit of mine, also seen in many code examples, and indicates an integer in a loop. You can replace it by anything you want, though.

In the examples above we have been using a for-loop on a sequence of numbers. The sequence also consists of anything else… and a list is an excellent way to store the content! Let's go back to our famous_neurosurgeons example and elaborate on it.

```
# Create list with famous neurosurgeons
famous_neurosurgeons = ['Cushing', 'Penfield', 'Leksell', 'Yasargil']

# Now iterate over this loop and print each single value
for neurosurgeon in famous_neurosurgeons:
print neurosurgeon
```

This is what makes lists so useful! Because if you would have stored each neurosurgeon in his/her own variable, this would not have been possible. As you can guess, it does not matter how many values you have in a list, the code to iterate over them remains as short as above.

So far we just printed the value, but let's elaborate on the BMI calculator too:

```
# Calculate BMI for 10 people

# Here we use a list with body mass and body length for 10 people
# Assume the order is the same in both lists
body_masses = [67, 88, 70, 73, 63, 81, 77, 59, 69, 76]
# in kg
body_lengths = [176, 190, 162, 155, 175, 169, 178, 165, 168, 174] # in cm

# Create empty list to store Body Mass Indexes
bmi_list = []

# Loop over masses and lengths, calculate BMI and add to list
for i in range(len (body_masses)):
    body_length_in_meter=body_lengths[i] / 100
    bmi = body_masses[i] / (body_length_in_meter **2)
    bmi_list.append(bmi)

# Print all values
print bmi_list
```

What did we do? Using the body masses and corresponding lengths of 10 people stored in a list, we looped over that list and for each item calculated the BMI (note that we first needed to convert length to meters) and then added it to the list. The result is a list that contains all the BMI values that we can now use for whatever we want. Again, the code to calculate the BMI values and add them to a list would have been the same if there would be 10,000 people in the original data. Rather powerful, these loops!

I used range(len(body_masses)) instead of range(10) to make the code more general. Although the second option may look easier and therefore less error-prone, it is not. What if you added a person to the list? In that case you also need to update the number in the range() function. The approach I used avoids this: If you add items to the list, the correct number is used automatically. So in the end, this code is less error-prone.

List comprehensions

A list comprehension is actually a shorthand notation that combines a list with a loop, and if you want, even with a conditional statement. Suppose you want to create a list with all numbers between 1 and 1000 that can be divided by 2 and by 3. Using the knowledge you have so far, you could do it like this:

```
# Standard way to solve this problem

results = []

for i in range(1000):
    if i % 2 == 0 and i % 3 == 0:
    results.append (i)

print results
```

If you would do that with a list comprehension, it looks like this:

```
# Same solution, less typing

results = [i for i in range(1000) if i % 2 == 0 and i % 3 == 0]
print results
```

Neat! Let's do another example:

```
# Again those famous neurosurgeons
famous_neurosurgeons = ['Cushing', 'Penfield', 'Leksell', 'Yasargil']

# Create a list of famous neurosurgeons with
# 'l' as the last letter of their name
last_letter_l = [ns for ns in famous_neurosurgeons if ns[-1] == 'l']
print last_letter_l
```

I admit the latter is not very useful, but that was not really the point of the example. I just wanted to show you can use list comprehensions with string values too. Note that because I won't be using the variable name ns outside this list comprehension, I chose ns instead of neurosurgeon (or something else that is more descriptive) just to save me some typing.

While loops

A while-loop is an alternative to the for-loop. You won't use it a lot, probably, but just to give you an idea:

```
# While-loop example

i = 1
```

```
while i <=10:
    print i
    i = i + 1
```

As you see, the iterator gets updated at the end instead of in the beginning. That means that first some code is executed and afterward you go to the next round of the loop. This can be useful if you do not know in advance how many iterations you will get. An example is a number guess game like this:

```
# Number guess game using while-loop

# This is the number that needs to be guessed
# In a real game you would like this to be
# a random number
solution = 36

# Set guess to a start value
guess = 0

# Now keep asking until the user guessed the correct answer
while guess != solution:

    # Ask user for a guess
    guess = int(raw_input('Enter a whole number between 0 and 100:'))
    print 'You guessed: %i' % guess

    # Compare guess with solution
    if guess > solution:
        print 'Too high!'
    elif guess < solution:
        print 'Too low!'
    else:
        print 'Congratulations, %i is the correct answer!' % guess
```

The while-loop keeps going until the users guessed the correct answer, independent of how many attempts this takes. And for here, that is exactly what we want!

ORGANIZING CODE

Functions

Congrats! You now know the basic elements of programming, so you should be able to build something useful already! The next step is to organize your code into reusable components. The most basic (and most common!) way to do so is by creating functions. You did already use some built-in functions, for example, len() and range(). Now, we are going to build one ourselves. Let's go back to our BMI example. You learned to calculate BMI like this:

```
# Calculate BMI

body_mass = 80
body_length = 180

bmi = body_mass/((body_length / 100) ** 2)
print bmi
```

This works fine, but every time you need this functionality you have to retype the formula. How inconvenient is that! Wouldn't it be better if we could use a function for this, like calculate_bmi()? Well, we can! But we have to define it first!

```
# We define a function to calculate BMI
def calculate_bmi (mass, length):
    result = mass/((length / 100) ** 2)
    return result

# Now we use the function to calculate bmi

bmi = calculate_bmi(80, 180)
print bmi
```

What happened here? First, we typed def to let the compiler know that we are going to define a function here. After that comes the function name, followed by parentheses. If the function takes any arguments (and most functions will do so), they are enclosed between the parentheses, and they are comma-separated. The function definition ends with a colon (:). On the next line, which is indented in the same manner as conditional statements and loops, we introduce a variable result to store the result of our calculation. It does not need to be named result, it can be named bmi, or strange_banana if you feel like it. Important is that this variable does only exist "inside" the function, hence it is called a local variable. If you were to ask for the value of result outside the function, you would get an error. At the last line, we return the result. The name return is mandatory if you want to return a value. You can also decide not to return a value, but simply to print something. The advantage of the return value is that is makes the code more reusable. Suppose we want to calculate the BMI and then use that value for some decision-making (if…else). This function will be very helpful. You should restrict a print output to functions that are just meant for printing some text to the screen.

Regarding your programming style, you could also do this:

```
def calculate_bmi(mass, length):
    return mass/((length / 100) ** 2)
```

This is up to you. For a single line of code, I prefer the latter, but if there is more going on, I will go for multiple lines and some local variables. Further, it is a good habit to include a short line of documentation about what the function does. Without going into details, the best way is between triple quotes (""") right below the function definition. This is not mandatory for functionality, though.

```
def calculate_bmi(mass, length):
    """Calculate Body Mass Index using body mass (kg) and body length (cm)"""
    return mass/((length / 100) ** 2)
```

Next step is to incorporate the decision-making. Let's add a function for that. What do you think about this example?

```
# Decision-making - the bad way
def calculate_bmi (mass, length):
    '''Calculate Body Mass Index using body mass (kg) and body length (cm)'''
    return mass/((length / 100) ** 2)

def dietary_advice (mass, length):
    '''Give dietary advice based on Body Mass Index'''
    bmi = mass/((length / 100) ** 2)

if bmi > 30:
    return 'No more burgers for you!!'

else:
    return 'All right, one burger will do.'

print dietary_advice(100, 180)
```

This works perfectly, but what I did is actually horrible! And no, I am not talking about the advice I gave. What if I later find out that I made a mistake in my BMI calculation formula, or I want to change something for a different reason? I would need to do that twice now! At some point, you will forget to do so, and get errors. There is an acronym DRY in programming which stands for Don't Repeat Yourself! So let's fix that right away!

```
# Decision-making - the good way

def calculate_bmi(mass, length):
    '''Calculate Body Mass Index using body mass (kg) and body length (cm)'''
    return mass/((length / 100) ** 2)

def dietary_advice(mass, length):
    '''Give dietary advice based on Body Mass Index'''
    bmi = calculate_bmi(mass, length)

if bmi > 30:
    return 'No more burgers for you!!'

else:
    return 'All right, one burger will do.'

print dietary_advice(100, 180)
```

Much better!! By the way, the local variables mass and length are not required to be the same for both functions. The solution below works too:

```
# Decision-making - another good way

def calculate_bmi (mass, length):
    '''Calculate Body Mass Index using body mass (kg) and body length (cm)'''
    return mass/((length / 100) ** 2)

def dietary_advice (body_mass, body_length):
    '''Give dietary advice based on Body Mass Index'''
    bmi = calculate_bmi (body_mass, body_length)

if bmi > 30:
return 'No more burgers for you!!'

else:
    return 'All right, one burger will do.'

print dietary_advice(100, 180)
```

What I do in the second (and third) example, is that I take the variables from the function definition and pass them to the calculate_bmi() function that I call from inside the dietary_advice() function. And that is not only possible, it is also DRY-compatible and is therefore recommended!

Now you know how to work with functions, lets continue to the final step of this crash course. You will learn how to save code into a file that you can run from the command line. That way, you can save your work and reuse it as often as you want.

Reusing code

In the section on calculations, we have been calculating tumor volume (assuming a spherical tumor) like this:

```
# Import math module
import math

# Ask for tumor diameter and calculate tumor volume
tumor_diameter = float(raw_input('Enter tumor diameter in cm:'))
tumor_volume = (4/3) * math.pi * (tumor_diameter / 2) ** 3

# Return tumor volume
print 'Tumor diameter: ' + str(tumor_diameter)
print 'Tumor volume: %. 2fcc' % tumor_volume
```

We are doing three things here. First, we import a math module that we need to work with pi (we did not simply use 3.14 for accuracy and educational reasons). Second, we ask for a tumor diameter and use that to calculate tumor volume. Third, we give feedback to the user.

Let's convert this into a neat solution that you could use in your professional life (if you feel like it). Although we could put the code above in a file and run it (which would work), we also have to consider that later on we may want to add functionality. For that reason, it is good to keep your code organized, and the best moment to start doing so is right from the beginning! Off we go…

```
# This little utility helps to calculate tumor volume
import math
def calculate_tumor_volume(diameter):
    '''Calculate tumor volume based on diameter'''
    return (4/3) * math.pi * (diameter / 2) ** 3

def print_tumor_stats (diameter):
    '''Print tumor parameters to the screen (based on diameter)'''
    print 'Tumor diameter: ' + str(diameter) + ' cm'
    print 'Tumor volume: %. 2f cc' % calculate_tumor_volume (diameter)

tumor_diameter = float(raw_input('Enter tumor diameter in cm:'))
print_tumor_stats(tumor_diameter)
```

Ah, much more pleasant for the eyes! First we defined some functions that make it easy to reuse the code. Then we simply

ask for the diameter and print the requested information. Run it and see that it works, before we continue!

One final remark is within the context of this crash course. All code examples you have been working with were running inside a code window in your browser. On ProgrammingForPhysicians.com, you will find a quick introduction on how you can install software locally to run the same code from your computer. This works on Windows, Mac OS X and Linux, and can be more practical in real life than running such code online. Feel free to have a look and experiment with it!

Where to go from here?

Congratulations, you really did master the basics of programming! Now where to go from here?

You learn to program by practicing, so if you want to expand your knowledge, that is what you should do. The web is full of information, especially on open-source languages. Just go to Google and type **python** followed by your search request. You will find it really quickly! And don't forget StackOverflow.com which is a very helpful resource.

On purpose I did not include graphical user interfaces. We have gotten used to them in all sorts of applications, but they are not the essence of programming. And to learn good programming, they are actually distracting!

Further, once you got the basics, do not feel restricted to Python. I really like this language for scientific computing and prefer to use it with the Anaconda distribution of IPython.[1] This includes a lot of additional modules that are very useful, and the IPython notebook format which recently even got attention of nature.[4]

If you want to go specifically for web development, I'd recommend taking a look at PHP. This is also open source and free, and many very affordable hosting providers on the web use the Apache-server which runs PHP (together with MySQL, an excellent database format). Facebook is based on PHP!

For mobile applications you can either choose for web-like interfaces that work on all platforms but do not offer a "native" look and feel (e.g., Corona, Cordova, Sencha, jQuery Mobile, and Telerik Platform) or native technologies. In the latter, you have to learn Swift for iOS development and Java for Android development.

The basics of programming do not change, just the syntax of the language changes. For example, our BMI example in Swift would look like this:

```
//Same code in Swift
//This could be used for an iPhone or iPad app

func calculateBMI(mass: Float, length: Float) -> Float {
    let cm = length/100
    return mass/(cm * cm)
}

func dietaryAdvice (bodyMass: Float, bodyLength: Float) -> String {
    let bmi = calculateBMI(bodyMass, bodyLength)

    if bmi > 30 {
        return "No more burgers for you!"
    } else {
        return "All right, one burger will do."
    }
}

print(dietaryAdvice (100, 180))
```

This code does exactly the same, but there are some differences in syntax. For example, a double forward slash is used for comments instead of a hashtag, Swift standards use camel-case notation, and variables (or actually, constants) are defined including the type of value they are going to contain. Further, indentation is not required in Swift (just recommended) and hence curly braces are used to organize pieces of code that belong together.

Having seen this example may not only help you to appreciate Python's simplicity but also to see that there are quite some similarities between the languages. Once you get the basic concept of programming, it is fairly easy to switch to another language, depending on your needs. The challenge is to know the language and platform specific features as far as you need them.

CONCLUSION

This crash course gave you an introduction in computer programming. Now, it is up to you to apply your medical knowledge and build new applications that will help both you and your colleagues all over the world in delivering better patient care! Have fun, and let's stay in touch.

REFERENCES

1. Download Anaconda. Available from: http://www.continuum.io/downloads. [Last accessed on 2016 Feb 12].
2. Kubben PL. NeuroMind. Available from: http://www.dign.eu/nm. [Last accessed on 2016 Feb 12].
3. Kubben PL. Why physicians might want to learn computer programming. Surg Neurol Int 2013;4:30.
4. Shen H. Interactive notebooks: Sharing the code. Nature 2014;515:151-2.
5. Welcome to Python.org. Available from: https://www.python.org. [Last accessed on 2016 Feb 12].