Python tutorial

Prof. Shahed Sharif

Contents

1 Getting started **1**

2 Basics of programming **1**
2.1 Putting the parts together 2

3 Basics of Python **2**
3.1 Arithmetic 3
3.2 Data 3
3.3 Booleans 5
3.4 Control flow 5

4 Functions and syntax **7**
4.1 Writing down the algorithm 7
4.2 Typing up the program 8

5 Good practices **9**
5.1 Start on paper 9
5.2 Write doc strings and comments 9
5.3 Use many smaller programs 9
5.4 Debug with tables 10

6 Rules for turning in programs **11**

7 Further reading **12**

8 Homework **12**

1 Getting started

Make sure you have Python 3 installed. It should come bundled with the IDLE application. But don't open it yet! Also, even if you don't finish today, make sure you read through to the end.

2 Basics of programming

What a computer program does—and by extension, how we understand computer programming—can be divided up into five parts:

I/O. This stands for "input/output". That is, the program will often ask for some information from the user (input), and at some point will display results (output). This can be very complicated if, for example, your program is a video game. For us, I/O will be very simple.

Arithmetic. Programs can do arithmetic. This means the 4 standard arithmetic operations.

Data. A program stores and manipulates data. Data is, broadly, stored in *variables*, of which there are a number of different types. A program can change the content of a variable, and also read the contents of a variable.

Booleans. A program can determine the truth or falsity of certain types of statements. A *Boolean value* is essentially either "true" or "false", and a *Boolean function* is something that inputs a bunch of statements and, based on the truth of those statements, outputs a Boolean value.

Control flow. Control flow refers to the order in which the computer executes commands. One way of looking at this is that control flow instructs the computer what it should do next. The most important part of control flow is *iteration* or *loops*; that is, doing the same or similar tasks or calculations repeatedly. Computers are much better suited to iteration than humans!

2.1 Putting the parts together

Any algorithm can be put together out of these constituent parts. When writing a program, your first task should be to figure out how to write your algorithm out of these parts. You should do this away from the computer!

3 Basics of Python

Okay, now you can start IDLE! When you do, you'll see a window on your screen which is called a "Python shell" or "Python interpreter". It's essentialy a glorified calculator. When you see stuff that looks like programming in this document (see below), type out each line *exactly as you see it* and press enter. If you get an error, it is overwhelmingly likely that you typed something incorrectly. Pay particular attention to tabs at the beginnings of lines—these are crucial to how Python functions!

When you get an output, make sure to understand why Python gave you that output.

3.1 Arithmetic

```
5+3
5-3
5*3
5*(2+3)
25/3
25//3
```

Note the difference between integer division and regular division!

There are some more useful operations; see if you can determine what they do:

```
2**3
17%5
```

Throughout, you'll see commands which I won't explain. Make sure to figure out what they do, and test your guesses with your own input!

3.2 Data

There are three basic data types we'll discuss: numbers, strings, and lists. As you've already seen, there are actually two types of numbers: integers and decimals (called "floats" in computer science). We'll mainly work with integers. A string is just a sequence of characters, such as a text message. We'll discuss lists in a bit.

```
x=5
x
print(x)
x+3
x=x+4
x
(x,y) = (3,5)
print(y,x)
x,y=y,x
print(y,x)
x,y = x+y, x-y
x, y
x='hi'
x
```

Note that = denotes *assignment*; that is, the expression on the right is evaluated, then assigned to the variable on the left. Also observe the commands `x,y = y,x` and `x,y = x+y, x-y`. Unlike in a lot of computer languages, in Python it is easy to swap variables or do multiple assignments without running into problems.

Strings are enclosed in quotes; either single or double-quotes are fine for simple cases. There are subtle differences between them which are relevant in more complex situations.

```
y=' there'
z=x+y
print(z)
print('z')
ord('Z')
ord('A')
chr(66)
chr(89)
```

Lists are finite sequences of other data types. You can think of them as vectors, except you can change the length of the list as much as you like. Elements of a list can be numbers, strings, or even other lists. Some of the commands below give errors; why?

```
li=[1,2,'apple']
li[0]
li[2]
li[3]
len(li)
li[len(li)]
li[len(li)-1]
li[1:2]
li[1:3]
li[1:]
li[:2]
li=li+['pear']
li
li=li[1:]
li
m=li+li
m
```

As you can see, list operations are very intuitive. Some list operations can be used with strings as well.

```
s='hell'
s=s+'o'
s
s[1]
'j'+s[1:]
```

There's much more to say about lists, and I'd strongly recommend doing some further reading on them. When in doubt, the input and output of your programs should be lists.

3.3 Booleans

We start with a single variable assignment ($x = 3$) and proceed with some truth tests and more complex boolean functions (such as AND and OR). Note the difference between = and ==.

```
x=3
x==3
x==2
x<3
x<4
x!=4
x!=3
x==3 and x<4
x==3 and x<2
x==3 or x<2
not (x==3 or x<2)
not x!=4
x='hi'
x+' there'=='hi there'
```

One handy trick that comes up when using control structures is that `0` and `[]` are equivalent to `False`, while all other values are considered `True`.

3.4 Control flow

Here's a simple loop:

```
for i in range(1,10):
    print(i)
```

When entering this into the interpreter, you need to hit enter on a blank line to let Python know you're done with the loop. (That is, after typing `print(i)`, hit enter twice.)

Observe the colon, indentation, and how the output relates to the bounds in the `range` declaration. IDLE should take care of the indentation for you, but only to a certain extent: to end the indentation, you have to press backspace on a new line.

What the loop does is start `i` at the value 1, execute the indented code, then increment `i` to the next value, 2. This continues until `i` hits the last value, 10, at which point the loop ends without executing the indented code. We call `i` the *index* for the loop.

Here's a useful variant:

```
for i in range(10):
    print(i)
```

In mathematical programs, we usually use an additional variable within a loop which calculates something. For example, if we wanted to compute $1 + 2 + \cdots + 9$, we could do it as follows:

```
sum = 0
for i in range(1,10):
    sum = sum + i
sum
```

(That last line could have been `print(sum)`.)

Another way of making loops is with **`while`**.

```
i, sum = 0, 0
while i<10:
    sum += i
sum
```

(Remember to hit enter twice after the last line!) Using **`while`** is almost always better than using **`for`**. The reason is that with **`for`**, you have to specify ahead of time how often the program will loop. With **`while`**, you get flexibility—the loop keeps going until the Boolean after **`while`** (in our case, `i<10`) evaluates to `false`. For this reason, the Boolean after **`while`** is called the *termination condition* for the loop.

Exercise 1. What happens if you do the same program, but with the two indented lines reversed? Guess first, then try it!

A common construction is to create, read, or change a list inside a loop. With a **`for`** loop, we usually use **`range`**`(0,`**`len`**`(`**`list`**`))` or something similar.

The last control structure is the **`if/else`** declaration.

```
if i%2==0:
    print(i, 'is even')
else:
    print(i, 'is odd')
```

The **`else`** is optional.

Exercise 2. Compute the 100th triangular number without the formula. Do it in two different ways: using a **`for`** loop and a **`while`** loop.

Exercise 3. Given that the third Fibonacci number is 2, compute the 30th Fibonacci number. (Hint: you will need more than one variable.)

Exercise 4. Create a list of the numbers 1 through 100; by "list", I mean the data type, so `[1, 2, ..., 100]`. Start with `li = []`.

4 Functions and syntax

We now figure out how to put everything together to write a program. As an example, we'll recreate multiplication of positive integers; that is, given positive integers `a` and `b`, we want to write a program which outputs the product of the two using only addition.

4.1 Writing down the algorithm

The key to writing a program is answering the questions

- "What are we looping?" and
- "How do we know when to stop looping?"

We should first do this on paper! We write down what we want in plain English:

1. Given $a, b \in \mathbb{N}$.
2. Write down a on a paper b times.
3. Add together.

Okay, not too bad. However, this doesn't answer either of our questions. For the first question, in step 2 we are looping adding a each time. For the second question, we stop once we've done b iterations. To implement the loop, we can use **`for`** or **`while`**; for this example, I will use **`while`**.

1. Given $a, b \in \mathbb{N}$.
2. Start a running total at 0.
   (a) Add a to our running total.
   (b) If we've done this b times, stop the loop.
   (c) If not, go back to (a).
3. Output the running total.

To make the loop work, we've introduced two new quantities: the running total, and the number of times we've been in the loop. The first quantity we can call `p` for product, while the second quantity is a counter related to `b`. The counter can start at 0 and work its way up to `b`, or can start at `b` and count down to 0. I will choose the latter, for reasons that will become apparent. When we do this, the termination condition is that the counter reaches 0.

4.2 Typing up the program

To type up the program, you *can* enter it into the same window you've been using this whole time, but you should not. Instead, in the File menu, click on "New window." This will open a different window which is essentially just a text editor, but one which is specialized to editing Python. Type out your program in this new window, save the file as a '.py' file, and then run it with the "Run module" command in the Run menu. The results will appear in the interpreter window. Note that all flow declarations end in a colon, and all instructions inside it are indented. The same formalism holds for functions, as you can see below.

```
def dumb_multiply(a,b):
    """Multiply positive integers a and b."""
    p = 0
    while b!=0:
        p,b = p+a, b-1
    return p
```

Note that our counter is `b` itself! To understand how the program works, I recommend choosing small values of input and try executing the program yourself, on a piece of paper (not with the computer).

Once you've typed the above and used "Run Module", you'll have defined the function `dumb_multiply` as a function of two arguments. In the interpreter you can use the function by entering, for example, `dumb_multiply(5,8)`. This looks just like a mathematical function, in this case of two variables.

Note that the program ends with a **`return`** statement. Run on its own, **`return`** acts just like **`print`**. However, **`return`** is superior because the output of **`return`** can be used as the input of another program. Indeed, complex programs are typically split into pieces, where one function will call the results of another. Note however that a **`return`** statement, unlike **`print`**, ends the enclosing function immediately.

Lastly, the variables `a` and `b` are *localized*, so even though their values are changed inside the function, they are not changed outside of the function. That is, if we did

```
a=18
b=15
dumb_multiply(a,b)
b
```

then the value of `b` is still 15, even though the value of `b` appears to change inside `dumb_multiply`.

Exercise 5. Write a program `str_to_num` that inputs a string (consisting only of capital letters) and outputs a list of number, 0–25, giving the numerical representation of the string, just as in the shift cipher. So

`str_to_num('ABC') = [0, 1, 2]` and `str_to_num('ZAP') = [25, 0, 15]`.

Exercise 6. Write a program `num_to_str` that does the reverse: on input a list of numbers (where each number is from 0–25), the program outputs a string of capital letters, each character corresponding to the appropriate number in the list. Thus

`num_to_str([0,1,2]) = 'ABC'` and `num_to_str([25, 0, 15] = 'ZAP'`.

Exercise 7. Write a program `num_shift` that on input a list of numbers 0–25 and a key k, adds k mod 26 to each number of the list. So

`num_shift([0, 1, 25], 3) = [3, 4, 2]` and
`num_shift([0, 1, 25], -49) = [3, 4, 2]`.

5 Good practices

5.1 Start on paper

This is important! *Never* start programming at your computer! Start at your physical, paper notebook. If you can't get it right on paper, there's no chance you'll get it right at the keyboard.

5.2 Write doc strings and comments

Look at the second line of the `dumb_multiply` function above. The text inside the triple quotes is called the *doc string* for the function. It provides a brief description of the function, and is always a good idea to include.

Any text beginning with the hash symbol # is considered a *comment*; that is, it is ignored by Python. This is good to explain your program to someone else reading your code, or more importantly, yourself in a few weeks when you're reusing the same program!

Here's the dumb multiplication function with lots of comments:

```
def dumb_multiply(a,b):
    """Multiply positive integers a and b."""
    p = 0 # p will be the product
    while b!=0: # loop by decrementing b
        p,b = p+a, b-1
    return p # we're done with the program. Champagne time!
```

5.3 Use many smaller programs

For long, complicated programs, break up the problem into several pieces and program them separately. As a simple example, suppose you want to input the lengths of two legs of a right triangle, then compute the sine of the smallest angle. You could do this in pieces as follows:

```
def hypotenuse(a,b):
    """Compute the hypotenuse of a right triangle.

    Legs have length a and b."""
    return (a**2 + b**2)**(0.5)

def triangle_sin(a,b):
    """Compute sine of smallest angle of right triangle.

    a, b are lengths of legs."""
    if a<b:
        return a/hypotenuse(a,b)
    else:
        return b/hypotenuse(a,b)
```

Exercise 8. Write a program `caesar` that on input a string and a key, outputs the appropriate Caesar shift. So

`caesar('HI', 1) = 'IJ'` and `caesar('ZAP', 2) = 'BCR'`.

5.4 Debug with tables

If your program isn't giving the right output and you're not sure why, try running the program "by hand." That means you should create a table of values for all the variables and then run the program yourself. An example is given in **Exercise 11.**

If you still can't figure out the problem, try inserting **`print`** statements inside your program, usually inside loops. Take the following example:

```
def dumb_multiply(a,b):
    """Multiply positive integers a and b."""
    p = 0
    while b!=0:
        p,b = p+b, b-1
    return p
```

This program will *not* correctly compute the product of two numbers. (Try it!) Not obvious why? Try doing it like this:

```
def dumb_multiply(a,b):
    """Multiply positive integers a and b."""
    p = 0
    while b!=0:
        p,b = p+b, b-1
        print a,b,p # This shows us what the program is doing
    return p
```

Then you'll see the values of the variables as the program iterates, and hopefully understand the problem.

Exercise 9. Write a program `gcd` that computes the gcd of two numbers. Hint: you will need two variables, `rold, rnew`. (Though you need not use those names!) Check that `gcd(128478,197351) = 7`.

Exercise 10. Write a program `egcd` that computes the Extended Euclidean algorithm; that is, `egcd(a,b) = [d, s, t]` where `d` $= \gcd($`a, b`$)$ and `sa` $+$ `tb` $=$ `d`. Check that `egcd(128478,197351) = (7, 7900, -5143)` (though your answer might be different from mine; if it is, check yourself that the last two numbers are correct.) Hint: I recommend having 6 variables in your program:

```
rold, rnew, sold, snew, told, tnew
```

(of course, you can give these different names). It can also be convenient to include a variable for `q`.

6 Rules for turning in programs

When turning in programs,

- do not use any external libraries—that is, anything with the **`import`** command;
- upload your python file (which should have a `.py` extension) to Gradescope;
- make sure code runs without errors, returns the correct results, and passes and assigned test;
- make sure all programs have doc strings;
- use previously written code where appropriate; and
- when required, include a proof that the code is correct with your written homework.

When you use previously written programs, clearly state what those programs do; specifically, the input and output of the programs. (You can include the code for earlier programs, but it isn't required.) Of course, that means that you should save your previous code, as it will be useful throughout the course.

I will demonstrate some proofs in class that various programs are correct. As for most math problems, there is no one-size-fits-all way to prove a program works, but there are general techniques. For example, induction is a fantastic tool for correctness proofs. Below is an example problem, based on the following code.

```
def sod(x):
    s=0
    while x > 0:
        s=s+(x%10)
        x=x//10
    return s
```

Exercise 11. Determine what the above program does **without a computer**, as follows.

(a) The values of `x,s` vary throughout the program. Given that we evaluate `sod(54132)`, construct a table with the values of `x` and `s`, where each row corresponds to the values as the program iterates.

(b) Since there is a **`while`**, it is not obvious that the program terminates—that is, that the program does not go into an infinite loop. Prove that, for any positive integer input `x`, the program does eventually terminate.

(c) Let x_n, s_n be the values of `x,s` after n iterations of the loop. For example, $s_0 = 0$. Determine what x_n and s_n are. (Normally, you would prove the claim by induction, but you do not have to do this here.)

(d) Using the previous parts, state what the program does, and prove your claim.

(e) What does `sod` stand for?

7 Further reading

This document is only meant to get you started. I encourage you to look online for more details on programming in Python! I'd especially suggest looking into lists (particularly *list comprehension*), printing, and the *dictionary* data type.

For many people, programming isn't really programming until you can make windows with buttons, drop-down menus, and other assorted widgets. That kind of programming, called *GUI programming*, is not necessary for this course, but if you're interested, the TKinter library is a good place to start.

8 Homework

As part of your homework due Friday, please turn in Exercises 3–10. For programs where sample outputs are given, please include those tests in your file (so for instance, `gcd(128478,197351) == 7`).