1 Getting started

Log into CoCalc and open up our course. There should be a folder called “Tutorial”, and inside of it two files; these are *jupyter notebook* files, and are the standard file type we will be using for our programming. Open up the “Practice” file. It may take some time to start up, so read up until §3 while you’re waiting.

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!
2.2 Writing programs in the notebook

In each cell, you can put a short, one-line command, or a long program. Once you’re ready to run your commands, press Shift-Enter from within the cell. The iPython notebook will run everything in the cell, top to bottom, and output whatever needs to be output. For example, type `print("Hello world!")` in the first cell and hit Shift-Enter. In most of the following, you should put each line in its own cell—that is, type a line, hit Shift-Enter, type the next line, etc. If nothing happens, there is not supposed to be output.

Note that in the “Cell” menu, there is a command to run all cells. This can be useful if you are resuming work after a break.

Warning: if the cell itself is selected, instead of the interior of the cell, typing will have different effects. For instance, typing “dd” while the cell is selected will delete the cell. You can tell what is selected by the color of the left margin of the cell.

3 Basics of Python

As explained above, type each Python line in a cell and execute it. Make sure to think about what is happening!

3.1 Arithmetic

<table>
<thead>
<tr>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5+3</td>
<td>8</td>
</tr>
<tr>
<td>5-3</td>
<td>2</td>
</tr>
<tr>
<td>5*3</td>
<td>15</td>
</tr>
<tr>
<td>5*(2+3)</td>
<td>25</td>
</tr>
<tr>
<td>25/3</td>
<td>8.333333333333333</td>
</tr>
<tr>
<td>25/3.0</td>
<td>8.333333333333333</td>
</tr>
<tr>
<td>int(25/3)</td>
<td>8</td>
</tr>
</tbody>
</table>

Note the difference between integer division and regular division!

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^3</td>
<td>8</td>
</tr>
<tr>
<td>17%5</td>
<td>2</td>
</tr>
</tbody>
</table>

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.

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

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.

```python
y=’there’
z=x+y
print(z)
print(’z’)
ord(’A’)
chr(65)
```

Lists are finite sequences of other data types. Elements of a list can be numbers, strings, or even other lists. Some of the commands below give errors; why?
```python
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. 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 \(==\).

```python
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:

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

The indentation is important. Note that CoCalc automatically puts an indentation in the second line. In Python, if a line ends in a colon, the next line should automatically be indented.

Observe the colon, indentation, and how the output relates to the bounds in the range declaration. CoCalc 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. Incidentally, \(i\) is called the index for the loop. In general, all indented lines are executed before incrementing the index; for instance, try

```python
for i in range(1,10):
    print(i)
    print(i^2)
```

Here’s a useful variant:

```python
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:
Math 523  
Spring 2023

Python tutorial

\[ \text{sum} = 0 \]
\[ \text{for } i \text{ in range}(1, 10): \]
\[ \quad \text{sum} = \text{sum} + i \]
\[ \text{sum} \]

(Use backspace to dedent. That last line could have been \texttt{print(sum)}.)

Another way of making loops is with \texttt{while}.

```
i = 0
while i < 10:
    print(i)
    i += 1
```

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

Using \texttt{while} is almost always better than using \texttt{for}. The reason is that with \texttt{for}, you have to specify ahead of time how often the program will loop. With \texttt{while}, you get flexibility—the loop keeps going until the Boolean after \texttt{while} (in our case, \texttt{i < 10}) evaluates to false. For this reason, the Boolean after \texttt{while} is called the **termination condition** for the loop. On the other hand, whenever a program has a \texttt{while}, there’s a possibility of an infinite loop; that is, the program goes on forever. For instance, if we delete the line \texttt{i += 1} from the above program, it runs forever. For such programs, proofs need to show that the program eventually terminates; that is, that the termination condition is fulfilled.

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

The last control structure is the \texttt{if/else} declaration.

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

The \texttt{else} is optional.

**Exercise 2.** Compute the 100th triangular number without the formula. Do it in two different ways: using a \texttt{for} loop and a \texttt{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.** Find the sum of all numbers between 1 and 40 which are not multiples of 3; that is, \[ 1 + 2 + 4 + 5 + \cdots + 40. \]

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 first step should be done 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 use any of the constructions of computer algorithms, specifically loops. Note that the second step is an iterative step; thus, we should use a loop. One can use \texttt{for} or \texttt{while}; for this example, I will use \texttt{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.
```
Notice that 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. The key observation is that the termination condition for the loop is that the counter, whatever we call it, equals 0.

4.2 Typing up the program

Type the entire program in its own cell. 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.

```python
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 executed the cell you'll have defined the function `dumb_multiply` as a function of two arguments. In later cells 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. You should always end a program with `return`, and will rarely if ever use `print`.

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

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

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

Go back to the Tutorial folder and now open up the second file, called Programs. Use the program stubs provided.

Exercise 5. Write a new version of `dumb_multiply` which uses a `for` loop.

Exercise 6. Write a program which computes \( n! \).

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 documentation string or just 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! Generally speaking, you probably won’t be commenting anything this way, except perhaps a variable where it might not be obvious what the variable represents.

Here's the dumb multiplication function with a comment:
def dumb_multiply(a,b):
    """Multiply positive integers a and b.""
    p = 0 # p will be the product
    while b!=0:
        p,b = p+a, b-1
    return p

5.3 Use only basic Python and previous programs

CoCalc actually runs Sage, a mathematical package based on Python. Sage has many built-in commands that will automatically do what you want. We want to avoid this in your programs. Therefore, in your programs, you are not allowed to use any Python except what is covered above. If you are not sure or want to check if I’ll make an exception, feel free to ask me. There is one major exception: you may use any code that you’ve previously written.

In fact, for long, complicated programs, it is useful to break up the problem into several pieces and program them separately; or perhaps even uses pieces that you’ve already programmed. 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."
    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 7. Write a program called primetest which inputs a positive integer and outputs True if the input is prime, False otherwise. Once you have written it, do primetest(17)==is_prime(17), and repeat with 25, 1741, and 1. In each case, you should get True.

Exercise 8. Write a program called primelist which inputs a positive integer n and outputs an ordered list of all primes p with 1 < p < n. As a test, primelist(n) == list(primes(1,n)) should return True for any positive integer n; try n = 1, 37, and 200.

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 9.

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
    print a,b,p # This shows us what the program is doing
    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.
6 Rules for turning in programs

You don’t have to do anything special to turn in your code; just make sure all of the relevant work is done in the assigned file in CoCalc. When completing programs for homework, your code should

• all be located in the assignment folder, ideally in the file labelled “HWn.ipynb”;
• use only commands covered in this tutorial, or specifically mentioned later;
• run without errors, in a reasonable amount of time, and returns the correct results;
• have appropriate documentation strings;
• use previously written code where appropriate; and
• if appropriate, include a proof that the code is correct.

As stated above, you may only use the programming commands from this Tutorial, or that I explicitly allow later on. Thus, for instance, doing something like

```python
def primetest(n):
    """Determine if n is prime.""
    return is_prime(n)
```

is not allowed! Especially the `import` command is not allowed.

If there’s some Python command you want to use that wasn’t covered in this Tutorial, feel free to ask; I may well allow it.

Sometimes I will give very large inputs. In order for your code to run in a reasonable amount of time (usually 10 seconds or less), you’ll need to use the appropriate theoretical tools. I may also mandate that a certain result be used in your code.

When you use previously written code, make sure to include that code in an earlier cell and make sure the code is properly documented (a docstring is usually enough).

If the problem states that an input has a particular form, you can assume it. For instance, if I say that the input of `primetest` is a positive integer, you don’t have to check if the input is negative.

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.

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

**Exercise 9.** Determine what the above program does without a computer, as follows. You will be asked to prove certain facts; do your best to come up with a rigorous argument.

(a) The values of $x, s$ vary throughout the program. Given that we evaluate $\text{sod}(54132)$, 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?

One tip on proofs: under the “View” menu in CoCalc, do “Toggle line numbers.” Then in the proof, you can refer to lines of code by line number; for instance, cell 5, line 2.
7 Additional exercises

Exercise 10. Write a program `smallestprimefact` that inputs an integer \( n \geq 2 \) and outputs the smallest prime factor of \( n \). Remember, your code does not have to check that the input is \( \geq 2 \); you can assume it.

Exercise 11. Write a program `allprimefact` that inputs an integer \( n \geq 2 \) and outputs a list of all prime factors, including multiplicity, of \( n \), from smallest to largest. For instance, `allprimefact(12) = [2, 2, 3]`. (Hint: use `smallestprimefact`.)

Exercise 12. Write a program `primexp` that inputs an integer \( n \geq 2 \) and a prime \( p \), then outputs \( \text{ord}_p(n) \). (Hint: use `allprimefact`.)

Exercise 13. Prove that the `egcd` program from class is correct.

The following are good practice if you are having trouble understanding how to program. You do not have to hand them in.

Exercise 14. Write a program that inputs a list of integers and outputs a list of just the prime integers in the list; so \([1, 2, 3, 4, 5]\) would yield \([2, 3, 5]\).

Exercise 15. Write a program that inputs a list and outputs the list backwards, so the input \([1, 2, 3]\) would yield \([3, 2, 1]\).

Exercise 16. Write a program that inputs two lists of the same length and interleaves them; so if the input were \([1, 2, 3]\) and \([4, 5, 6]\), then the output would be \([1, 4, 2, 5, 3, 6]\).

Exercise 17. Write a program that inputs two lists of the same length and interleaves the first with the reverse of the second; so if the input were \([1, 2, 3]\) and \([4, 5, 6]\), then the output would be \([1, 6, 2, 5, 3, 4]\).