Translating to Python

Step Five: Translating to Code

Now that you are confident in your algorithm, it is time to translate it into code. This task is something that you can do with pencil and paper (as you often will need to do in a programming class on exams), but most of the time, you will want to actually type your code into an editor so that you can compile and run your program. Here, we will primarily focus on the mechanics of the translation from algorithmic steps to code. We **strongly** recommend that you acquaint yourself with a programmer's editor (Emacs or Vim) and use it whenever you program. We already covered Emacs, which we recommend that you use.

Function Declaration

We should start Step 5 by writing down the declaration of the function that we are writing, with its body (the code inside of it) replaced by the generalized algorithm from Step 3, written as *comments*. Comments are lines in a program which have a syntactic indication that they are for humans only (to make notes on how things work, and help people read and understand your code), and not an actual part of the behavior of the program. When you execute code by hand, you should simply skip over comments, as they have no effect. In Python, comments are written with a # sign. Anything after the # sign (to the end of the line) is considered a comment and is ignored by the interpreter.

One thing we may need to do in writing down the function declaration is to figure out its parameter types and return type. Even though Python is loosely typed, we should still think carefully about these types and know what they are. While we do not need to declare them explicitly in the code, we should still write comments that explain what types are used, so that we can understand this easily later when making use of the code. These types may be given to us—in a class programming problem, you may be told as part of the assignment description and in a professional setting, it may be part of an interface agreed upon between members of the project team—however, if you do not know, you need to figure this out before proceeding.

Returning to our rectangle intersection example, we know that the function we are writing takes two rectangles, and returns a rectangle. Earlier, we decided that a rectangle could be represented as four numbers. We could pass four numeric parameters for each rectangle (8 in total for 2 rectangles). Such a function call is cumbersome. We might also pass a tuple, which groups multiple items together. You will learn more about tuples and lists later on. Another option, which we will learn about later is to make a **class** for a rectangle, and then pass a rectangle object. You will also learn about object oriented programming later in this course.

In most other programming languages, we would need to think about what type of number we need—an integer, or a real number—and possibly how many bits we need for that integer or real number. Python does distinguish between integers and real numbers (called floats), but due to the loose typing we often do not need to distinguish explicitly. The answer to the question about which type of number we need is "It depends." You may be surprised to learn that "It depends" is often a perfectly valid answer to many questions related to programming, however, if you give this answer, you should describe what it depends on, and what the answer is under various circumstances.

For our rectangle example, the type that we need depends on what we are doing with the rectangles. Real number (floats) would make sense if we are writing a math-related program where our rectangles can have fractional coordinates. If we are doing computer graphics, and working in the coordinates of the screen (which come in discrete pixels), then an integer (int) makes the most sense, as you cannot have fractional pixels.

With this decision made, we would start our translation to code by declaring the function and writing the algorithm in comments.

Translating Algorithm Components

Once you have written the function declaration and written the algorithm steps as comments, you are ready to translate each step of the algorithm to code, line by line. If you have written good (i.e., clear and precise) steps in Step 3, this translation should be fairly straightforward—most steps you will want to implement naturally translate into the syntax we have already learned.

Repetition

Whenever you have discovered repetition while generalizing your algorithm, it translates into a loop. Typically, if your repetition involves counting, you will use a ***for*** loop. For loops are by far the most common loop in Python since you can use them to not only count, but iterate over items in a collection, such as a list, set, or dictionary. We will learn more about lists, sets, and dictionaries soon. If your repetition is neither counting nor doing something for each item in some collection, you should use a **while** loop. A while loop lets you check some conditional expression, then perform the steps in the loop if the condition is true. At the end of the loop, execution returns to the top, where the condition is checked again. Accordingly, you can repeat steps based on whatever conditional expression you have determined is appropriate.

Decision Making

Whenever your algorithm calls for you to make a decision, that will translate into either **if/else**. If you have many choices, you can use **elif** which is short for else-if.

Math

Generally, when your algorithm calls for mathematical computations, these translate directly into expressions in your program which compute that math.

Names

When your algorithm names a value and manipulates it, that translates into a variable in your program. You need to think about what type the variable has, and declare it before you use it. Be sure to initialize your variable by assigning a value to it before you use it—which your algorithm should do anyways (if not, what value did you use when testing it in Step 4?).

Altering Values

Whenever your algorithm manipulates the values that it works with, these translate into assignment statements—you are changing the value of the corresponding variable.

The answer is...

When your algorithm knows the answer and has no more work to do, you should write a ***return*** statement, which returns the answer that you have computed.

Complicated Steps

Whenever you have a complex line in your algorithm—something that you cannot translate directly into a few lines of code—you should call another function to perform the work of that step. In some cases, this function will already exist—either because you (or some member of your programming team) has already written it, or because it exists in the Python library. In this case, you can call the existing function (possibly reading its documentation to find its exact arguments), and move on to translating the next line of your algorithm.

In other cases, there will not already be a function to do what you need. In these cases, you should decide what arguments the function takes, what its exact behavior is, and what you want to call it. Write this information down (either on paper, or in comments elsewhere in your source code), but do not worry about defining the function yet. Instead, just call the function you will write in the future and move on to translating the next line of your algorithm. When you finish writing the code for this algorithm, you will go implement the function you just called—this is a programming problem all of its own, so you will go through all of the Steps for it.

Abstracting code out into a separate function has another advantage—you can reuse that function to solve other problems later. As you write other code, you may find that you need to perform the same tasks that you already did in earlier programming problems. If you pulled the code for these tasks into their own functions, you can simply call those functions. Copy/pasting code is generally a terrible idea—whenever you find yourself inclined to do so, you should instead find a logical way to abstract it out into a function and call that function from the places where you need that functionality.

With a clearly defined algorithm, the translation to code should proceed in a fairly straightforward manner. Initially, you may need to look up the syntax of various statements (you did make that quick reference sheet we previously recommended, right?), but you should quickly become familiar with them. If you find yourself struggling with this translation, it likely either means that your description of your algorithm is too vague (in which case, you need to go back to it, think about what precisely you meant, and refine it), or that the pieces of your algorithm are complex and you are getting hung up on them, rather than calling a function (as described above) to do that piece, which you will write afterwards.

Ending Blocks with Pass

Unlike most programming languages, Python uses (only) indentation to delimit blocks of code. Something is inside of an if or while if it is indented over, and returning to the outer indent level ends the block.   For example,

```
if x < 42:
    f()
g()
```

Here f() is inside of the if (only done if x<42) and g() is outside of the if (always done). By contrast, most other programming languages use explicit syntax for blocks. For example, in C (or C++, or Java) the above code would be:

```
if (x<42){
    f();
}
g();
```

Here, the curly braces ("{...}") explicitly delimit the block. We know the block ends when we reach the close curly brace.

Having an explicit end to a block is nice to help our reader (whether human or our editor) understand our intentions. In general, when we should prefer to be explicit wherever possible, as it enhances clarity and reduces the likelihood of introducing and error.

In Python we can indicate our explicit intention to end a block by the convention of writing "pass". The "pass" keyword means "do nothing" in terms of program effects. However, by convention it indicates "end of a block with nothing else to explicitly indicate its end". That is, if we write:

```
if x < 42:
    f()
    pass
g()
```

the "pass" statement on line 3 says (by convention) "I intended to end this block here". Any reasonable editor will understand this convention and will automatically go to the next outer indentation level after you type pass (and hit enter). We strongly recommend that you use this convention in your Python coding.

We note that there are cases where the block is explicitly ended by the nature of what is written, in which case, you should not write pass. Consider this code:

```
if x < 42:
    y = f()
    return y + 2   # explicitly ends block anyways, no need for pass
g()

if z > q:
    x = 3   # since we have the else that goes with the if, no need for pass
else:
    x = q-z
    pass  # pass here is good!
```

Here, we have two blocks that don't need a pass. *return* explicitly ends a block (it doesn't make sense to write anything after it inside the *if*, since such a statement would never be executed). Likewise, the *else* that goes with the *if* explicitly indicates the end of the *if*'s body.

Top Down Design and Composability

The process of taking large, complex pieces, and separating them out into their own function—known as ***top-down design***—is crucial as you write larger and larger programs. Initially, we will write individual functions to serve a small, simple purpose—we might write one or two additional functions to implement a complex step. However, as your programming skill expands, you will write larger, more complex programs. Here, you may end up writing dozens of functions—solving progressively smaller problems until you reach a piece small enough that you do not need to break it down any further. While it may seem advantageous to just write everything in one giant function, such an approach not only makes the programming more difficult, but also tends to result in a complex mess that is difficult to test and debug. Whenever you have a chance to pull a well-defined logical piece of your program out into its own function, you should consider this an opportunity, not a burden. We will talk much more about writing larger programs later on after you master the basic programming concepts and are ready to write significant pieces of code.

Composability

When you are translating your code from your algorithmic description to Python (or whatever other language you want), you can translate an instruction into code in the same way, no matter what other steps it is near, or what conditions or repetitions it is inside of. That is, you do not have to do anything special to write a loop inside of another loop, nor to write a conditional statement inside of a loop—you can just put the pieces together and they work as expected.

The ability to put things together and have them work as expected is called ***composability*** and is important to building not only programs, but other complex systems. If you put a for loop inside of an if statement, you do not need to worry about any special rules or odd behaviors: you only need to know how a for loop and an if statement work, and you can reason about the behavior of their combination.

In general, modern programming languages are designed so that features and language constructs can be composed, and work as expected. Python follows this principle pretty well, so you can compose pretty much anything you learn with pretty much anything else.

Stars Example

Now that we have seen how to translate our generalized steps into code, we can finish our earlier example of the "triangle of stars" problem.

We might start with the following algorithm, as comments in our Python code:

```
 def print_star_triangle(n):
     #  Count (call it i) from 1 to n (inclusive)
         #  Print i stars
         #  Print a newline
```

Notice that "print i stars" is a fairly complex step, so we should abstract it out into its own function and repeat the 7 steps on that function first.

We can come up with the following algorithm for print_n_stars:

```
def print_n_stars(n):
     #  Count from 1 to n  (inclusive of both)
         #  Print a star (with no newline)
```

For this function, we need to translate two lines of code. The first is counting, so it translates into a for loop.

The second is a print statement, so we would print('*'). However, if we just do *print('*')*, Python will automatically add a newline, so we need to tell Python to skip the newline with *end=''*

```
def print_n_stars(n):
      #  Count from 1 to n  (inclusive of both)
      for i in range(1, n + 1):
          #  Print a star (with no newline)
          print('*', end='')
          pass
      pass
```

We have written "pass" at the end of each indented block as there is nothing that explicitly ends that block. *pass* does nothing, but serves to tell our editor (and others reading the code) that we intended to end the block there. It is a convention, and not required. However, we HIGHLY recommend that you adopt it.

With print_n_stars written, we can run and test that function before we proceed. The more time you spend testing, the LESS time you spend debugging. The sooner you catch a bug the better! If there is a bug in print_n_stars, and you can find it now, you can save yourself time later. If you do not find it now (and go on to write print_star_triangle, then you will not know if a bug is in print_n_stars or print_star_triangle, and it will be harder to debug. If you test and debug now, then you can be confident in print_n_stars, and limit where other bugs could be. While this may seem trivial for such small functions, getting into the habit of incremental testing is critical as you move to larger functions.

Returning to print_star_triangle, we can now just call a function for the "print i stars" step. The other two steps are to count (a for loop) and to print a newline (a call to print with no arguments passed).

Here is the code for all of print_star_triangle:

```
def print_n_stars(n):
      #  Count from 1 to n  (inclusive of both)
      for i in range(1, n + 1):
          #  Print a star (with no newline)
          print('*', end='')
          pass
      pass


 def print_star_triangle(n):
     #  Count (call it i) from 1 to n (inclusive)
     for i in range(1, n + 1):
         #  Print i stars
         print_n_stars(i)
         #  Print a newline
         print()
         pass
     pass
```

Day 4, Part 2

- ☑ Step 5 Knowledge Check
  Please complete this Quiz to check your knowledge of Step 5 concepts