1 A universal Python program

In Topic 1, we've already seen how one computer program can analyze other computer programs. This kind of interplay between computer programs can be pushed even further: it's actually possible for one computer program to *execute* another computer program. To modern computer users, this is not surprising at all. In fact, many of the everyday operations we perform on a computer involve one program executing another program. Examples include:

- **Running a Python program from the command line:** When you enter the command "`python someProgram.py`" at a command prompt, you are using one program (the Python interpreter, which on my Windows computer just happens to be the program `C:\Python27\python.exe`) to execute another program (the Python program `someProgram.py`).
- **Launching Microsoft Word:** Suppose you double-click on a Microsoft Word document, thus launching the program Microsoft Word (and also opening the document). In this case, you have used one program (your operating system, for example OS X on an Apple computer) to launch another program (Microsoft Word, which happens to be stored as a file actually called `Microsoft Word` in typical Microsoft Office installations on Apple machines).

These examples are so mundane and obvious that it hardly seems worth discussing them. But I claim that even these everyday examples have an almost magical aspect to them: isn't it rather wonderful that the Python interpreter seems to be capable of executing *any* legal Python program? Similarly, the fact that your operating system can launch and run any correctly-written application program is immensely powerful. Arguably, this capability lies at the heart of the desktop computer revolution that began in the 1980s and continues to this day with smartphones, tablets, and other computing devices. Perhaps the most wonderful thing about computers is that they can run any program you want—not just a fixed program or programs preinstalled by the manufacturer.

But let's push a bit harder on this concept. In both examples above, we have a special type of program (e.g. the Python interpreter) executing a specific *different* type of other programs (e.g. source code written in the Python language). Can we come up with an example of a program of a particular type, that executes other programs of that *same* type? The answer is yes. In fact, it turns out that any programming language (with a minor caveat, discussed later) can be made to execute other programs written in that same language. So it's possible, for example, to write a Java program that can execute other Java programs. But some languages—including Lisp and Python—make this particularly

easy, by building in a self-execution facility. In Python, for example, there is an `exec` statement that does exactly this. Thus we can write

```
command = "print 'abc', 5+2"
exec command
```

When executed, this code produces the output "`abc 7`". But of course, `exec` can be used on arbitrarily complex strings, including entire Python programs. This leads us to our first example of what is called a *universal* program: `universalSimple.py`, shown in Figure 1.

```
# Python program universalSimple.py
exec input
```

Figure 1: The Python program `universalSimple.py`.

Try it now: at a command prompt, enter "`python universalSimple.py yes.py`". Another simple test would be "`python universalSimple.py redDevils.py`". In each case, the output is identical to running the input program (`yes.py` or `redDevils.py`) on its own. That is, we could produce the same effect with the commands "`python yes.py`" and "`python redDevils.py`".

Unfortunately, `universalSimple.py` is a little *too* simple for our purposes. You probably already noticed that it cannot deal with programs that themselves require inputs. For example, we can't emulate the command "`python containsZ.py someFile.txt`" using `universalSimple.py`. But it's not hard to fix this, producing a superior universal program called `universal.py`. We just need to agree on a few conventions for the input to `universal.py`. To start with, just as in Topic 1, we will need to combine program source and program input into a single string, separated by an agreed separator such as "`##end-of-program##`". Also, let's agree to always use the function form of a given program—this isn't essential, but it will avoid having to mess around with temporary input and output files for the program to be executed. Finally, we need a simple way of determining the name of the function to be executed. To do this, let's agree to name the file according to following recipe: the name of the function, concatenated with "`WithInput.txt`". For example, to execute the program `containsZ.py` on the input "`asdfZ4321`", we create the file `containsZWithInput.txt` shown in Figure 2. Now we can run the command

```
python universal.py containsZWithInput.txt
```

achieving exactly the same effect as running `containsZ.py` with input "`asdfZ4321`".

The program `universal.py` itself is shown in Figure 3. There are some slightly fiddly aspects involved in (i) splitting the input string into a program and a new input string for that program (lines 3–8); (ii) extracting the name of the function we need to call (lines 10–12); and (iii) executing the function definition so it is available in the remainder of the

```
# file containsZWithInput.txt
def containsZ(input):
    if "Z" in input:
        return "yes"
    else:
        return "no"
##end-of-program##
asdfZ4321
```

Figure 2: Combined program and input string `containsZWithInput.txt`, to be used as input for `universal.py`.

Python program (line 17). However, the key part of this universal program is very simple: in the final three lines, the program constructs a string such as

"`output = containsZ(inputToProgram)`",

executes this string, and prints the value of the resulting output variable.

You may be wondering how the Python `exec` statement actually works. We won't be delving into any details here, but if you think about it the right way, it's not at all surprising that Python can provide this functionality. Consider the specific example of our command "`python universal.py containsZWithInput.txt`". This invokes the Python interpreter, which begins executing `universal.py`. When the interpreter reaches line 22, the `exec` statement presents the interpreter with a brand-new piece of source code: the string "`output = containsZ(inputToProgram)`". But of course, this is precisely the kind of task that the Python interpreter was designed for, so it can continue as normal. The only difference is that it now interprets a piece of code that was generated dynamically, rather than interpreting static source code (i.e. code that was already in a source file before the program started running).

2 Universal Turing machines

So, it's clear that universal Python programs exist. What about Turing machines? It will come as no surprise that it's also possible to construct universal Turing machines. In fact, Alan Turing constructed a universal Turing machine in his 1936 paper which first introduced the concept of (what we now call) Turing machines. But Turing's construction is notoriously difficult to understand, and even contained a few mistakes.

Fortunately, we don't need to understand the details of this construction, or any of the universal Turing machines that were subsequently developed by other scholars. There is an easier way, based on the equivalence of Python programs and Turing machines. Recall from

```
# Python program universal.py

# Split input into the program to be executed,
# and the input to be given to that program
separator = "##end-of-program##"
splitInput = input.split(separator,1) # splits on first occurrence only
programToExecute = splitInput[0]
inputToProgram = splitInput[1]

# Determine the name of the function that needs to be executed.
suffix = "WithInput.txt"
functionName = inputFileName[:-len(suffix)]

# The next line does *not* invoke the function. Instead,
# it executes the *definition* of the function, which makes that
# definition available for later.
exec programToExecute

# The desired function is now defined, so we can execute it
# and print the output.
command = "output = " + functionName + "(inputToProgram)"
exec command
print output
```

Figure 3: A universal python program: `universal.py`.

the last lecture that it's easy to write a Python program which simulates Turing machines. (This was suggested as an optional exercise; we didn't actually prove it.) We do, of course, need to agree on an alphabet, and on notation for describing Turing machines and their inputs. But once this is done, we can easily write `simulateTM.py`: a Python program that takes as input the concatenation of (i) the description of some Turing machine, and (ii) the input to be given to the Turing machine. If and when the simulation halts, `simulateTM.py` outputs the current contents of the machine's tape, together with a "`yes`" or "`no`" indicating whether the machine halted in the accepting state.

But in the last lecture we also sketched a proof that, given any Python program, there exists a vanilla Turing machine that performs the same computation. Therefore, there exists some Turing machine U that mimics `simulateTM.py`. And by construction, U is a universal Turing machine.