tu06_re_IfsLoopsFunctions

February 2, 2023

1 Ifs, loops, and functions review

Last time, we reviewed the basic nouns of Python - objects and their types.

Here, we will review dome of the basic verbs (program structures and operators) that operate on the objects. Together, these things make our programs interesting and useful.

1.1 Conditional execution

One of the key things that makes computer programs really useful is the ability to do different things under different conditions. This is also referred to as control flow (you control the flow of your program rather than it running strictly line-by-line) or branching (your program can "branch off" to different code depending on conditions - *this does not refer to git branching*)

As we are talking about computers here, note that all decisions are ultimately going to depend on a Boolean value; if the value is `True`, we do something, if the value is `False`, we do something else.

1.1.1 Choosing with `if`

The absolute workhorse for control flow in Python is the `if()` statement. Let's remind ourselves what a simple program using an `if` looks like.

[1]:
```
nm = input("What's your name?")

if len(nm) < 4 :
    print(f'Hi {nm}! That is a short name!')
else :
    print(f'Hi {nm}!')
```

```
What's your name?Jay
Hi Jay! That is a short name!
```

Here there is one test, the `len(nm)` following `if`. If it's `True`, we run the code beneath. If it's `False`, then we run the code beneath `else`, the ever-loyal companion to `if`.

An `if` doesn't always need an `else`. For example, if you wanted to use rounded numbers (integers) only in a calculation, you could round the number to and `int` if you were given a `float`, but otherwise do nothing. In the cell below, write code to do just this using a lonely `if`

[5]:
```
a_num = 1 # change this to test your code

```

```
# your code here

num_type = type(a_num)
print(f'The final number is {a_num} of type {num_type}')
```

```
The final number is 1 of type <class 'int'>
```

Another companion to `if` is `elif`, a Middle Earth contraction of 'else if'. It's used identically to `if`, except that it always follows an `if` or another `elif`.

In the cell below, add an `elif` between the `if` and the `else` to modify the code so that it separately greets those with long names!

[8]:
```
nm = input("What's your name?")

if len(nm) < 4 :
    print(f'Hi {nm}! That is a short name!')
elif len(nm) == 7:
    print('Hi', nm, 'you name have 7 letters.')
else :
    print(f'Hi {nm}!')
```

```
What's your name?jedsada
Hi jedsada you name have 7 letters.
```

1.1.2 trying with `try`

Because Python is a very Zen language, you might think that it has a lot in common with the Jedi ways of Yoda. Perhaps. In Python, however, there ***is*** a `try`.

The `try` is used to run code that might fail. You could argue that "code that might fail" is the same as "all code", and you would have a very good point! In fact, some people write virtually all their code using a `try` (or its equivalent in other languages) at the very top. So their code looks like

```
try :
    the entire computer program
except:
    print errors
    print values of variables
    close any open files
    etc.
```

Normally, when Python encounters an error, it exits your code unceremoniously and prints the error with some angry red text.

In the cell below, try dividing some number by 0.

[9]:
```
3/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Input In [9], in <cell line: 1>()
----> 1 3/0

ZeroDivisionError: division by zero
```

In this case, we do get a clear diagnostic error – wish that were always the case! – but we get yelled at with red text and kicked out of Python. Who needs that kind of blow to their self-esteem?

We can use `try` to handle the error (or "exception") within our code so Python doesn't kick us out, and to give the user more polite and descriptive error messages.

Here's an example. Run it and enter a non-zero value, and then re-run it entering zero ("0" not "zero").

[10]:
```
denom = input('Please enter a denominator to divide into 3')
denom = float(denom)

try :
    answer = 3/denom
    print(f'3 divided by {denom} is {answer}!')
except :
    print('Oops, sorry, something went wrong - did you enter a zero?')
```

```
Please enter a denominator to divide into 30
Oops, sorry, something went wrong - did you enter a zero?
```

Note that, in this case, Python didn't actually kick us out. You can prove this by printing something below the `try&except` blocks. Like `print('See? Still running!')` or whatever. Use the cell below to try this.

[11]:
```
denom = input('Please enter a denominator to divide into 3')
denom = float(denom)

try :
    answer = 3/denom
    print(f'3 divided by {denom} is {answer}!')
except :
    print('Oops, sorry, something went wrong - did you enter a zero?')

print('It is still running.')
# print something here
```

```
Please enter a denominator to divide into 30
Oops, sorry, something went wrong - did you enter a zero?
It is still running.
```

Now try actually entering "zero" in the above. What happens? Why?

- The code is still running despite making it calculate 3/0.

You can also catch specific types of exceptions to help out even more. Here's an example:

[13]:
```
an_index = input(f'Pick a number from 1 to 4!')
an_index = int(an_index) - 1 # turn people index to Python index

denom_list = [3, 6, 0, 1]

try :
    divisor = denom_list[an_index]
    answer = 3/divisor
    print(f'3 divided by {divisor} is {answer}!')
except ZeroDivisionError as divzero_err :
    print('Oops, sorry, something went wrong - did you enter a zero?',␣
 ↪divzero_err)
except IndexError as ind_err :
    print('Mmm, looks like your number wasn\'n 1-4.', ind_err)
except :
    print('Oops, sorry, something went wrong and I\'m not sure what.')
```

```
Pick a number from 1 to 4!2
3 divided by 6 is 0.5!
```

Try (ha) this with some in-bounds and out-of-bounds numbers to verify it's working!

A few useful exceptions are:

```
* `IndexError` - an index is out of bounds.
* `KeyError` - an undefined dictionary key was used
* `NameError` - an undefined variable name was used
* `TypeError` - an operation was tried on an inappropriate type
```

In the cell below, write a little program that catches either a name or type error (or both).

[34]:
```
name = input('Write \'Jay\': ')

if(name.lower() == 'jay'):
    print('Yes, that is the correct spelling.')
else:
    print('No, that is not the correct spelling')
```

```
Write 'Jay': jay
Yes, that is the correct spelling.
```

1.2 Repeated execution

Repeating a calculation, or "looping" (with different input of course) is absolutely fundamental to computing. In fact, the very first "modern" computer was built to simulate atomic explosions with different initial conditions in order to build the first atomic bombs. In this case, and in general, it

is more practical and less time consuming to try things, whether by trial-and-error or a systematic search, with computer simulations than it is in real life.

Repeated execution can be more mundane but useful nevertheless. For example, you might have data file from each of 2000 participants in an online experiment, and you need to extract some data from each file, compute the mean and get the age for each person, and then plot the means by age. Would you rather

- manually open each file, getting the age and the data, and computing the mean

or

- have some code automatically chew through the files, compute each mean, and store the means and ages?

If you know how to write a little code then this is, as they say, the easiest decision in the history of decisions.

The looping functions in Python are `for` and `while`. A `for` loop executes code *for* a specific number of times, whereas `while` executes code *while* some condition is `True`.

1.2.1 Repeating a number of times with `for`

We use a `for` loop under two circumstances:

- we know how many times we need to do something
- we have an *iterable* that we need to go through

Looping over a range of values In the first case, we'll most often use the `range` function to provide us with the indexes for our loop. Like this.

[13]:
```
for i in range(5) :
    print(i)
```

```
0
1
2
3
4
```

You can also give range two arguments, which become the starting and stopping points, or three, where the third is the step.

[14]:
```
for i in range(1, 11, 3) :
    print(i)
```

```
1
4
7
10
```

In the cell below, write a `for` loop that prints the even numbers from 2 to 10.

[16]:
```
for i in range(2,11,2):
    print(i)
```

```
2
4
6
8
10
```

We can put it in reverse and go backwards as well! Run the code below!

[17]:
```
# countdown!
for i in reversed(range(1,11)) :
    print(f't minus {i} iterations to lift off!')

print('')
print('    |')
print('   / \\')
print('   | |')
print('   | |    LIFTOFF!')
print('   | |')
print('  / | \\')
print('')
```

```
t minus 10 iterations to lift off!
t minus 9 iterations to lift off!
t minus 8 iterations to lift off!
t minus 7 iterations to lift off!
t minus 6 iterations to lift off!
t minus 5 iterations to lift off!
t minus 4 iterations to lift off!
t minus 3 iterations to lift off!
t minus 2 iterations to lift off!
t minus 1 iterations to lift off!

    |
   / \
   | |
   | |    LIFTOFF!
   | |
  / | \
```

The little rocket above is an example of a "BUAG" or Butt Ugly ASCII Graphic. ASCII is "American Standard Code for Information Interchange" and is, basically, the symbols you can make with your keyboard. When a terminal was all we had, this was the only way to make pictures!

Quick Quiz! : Why does the right side of the rocket above have double backslashes in two of the

`print()` statements?

- Because without the second \ the code will not consider ' closing statement.

Each ASCII character has a code, and each code refers to one and only one character. The `ord()` function tells us the code for a symbol, and the `chr()` function goes the other way. Play around with the code below to see some characters and their codes.

[18]:
```
print(ord('A'))
print(chr(97))
print(ord('$'))
```

```
65
a
36
```

Hey, now we can use 36 as slang for "money"!

When you are using a loop, it's often handy to use the `.append()` method to store things computed within the loop. As an example, let's compute and store the first 10 values of the binary sequence.

[19]:
```
bin_seq = [] # make an empty list

for i in range(1, 11) :
    bin_seq.append(2**i)

bin_seq
```

[19]:
```
[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

(The binary sequence tells us that 1 bit can store 2 values, 2 bits can store 4 values, etc.)

Looping over an iterable One of the fantastic things about Python is that all the multi-element data types, like lists, are capable of providing their values one-at-a-time to a `for` loop. Like this – run the code below:

[35]:
```
a_list = ['a', 'b', 'c']

for i in a_list :
    print(i)
```

```
a
b
c
```

You can also iterate over a `tuple` or even a `string`. So we could have written the above code like this (run it to confirm):

[36]:
```
a_str = 'abc'

for i in a_str :
    print(i)
```

```
a
b
c
```

Let's see if we can take the alphabet and flip it backwards using a `for` loop. To do this, we'll use the `reversed()` function and a `for` loop to create a list with the elements (letters) of the alphabet in reversed order, and then, we'll use the `join` method to convert our list to a string.

Take look through this code line-by-line. Then run it and see if it works!

[37]:
```
alpha = 'abcdefghijklmnopqrstuvwxyz' # now I know my abc's...
separator = '' # the string (empty in this case) to separate our list elements␣
 ↪when using join
backwards_alpha = [] # empty list to hold our reversed letters

for i in reversed(alpha) :    # loop through the alpha string backwards
    backwards_alpha.append(i) # add each letter to our list in turn

backwards_alpha = separator.join(backwards_alpha) # make a string from our list␣
 ↪elements

print(backwards_alpha) # show off our work
```

```
zyxwvutsrqponmlkjihgfedcba
```

Memorize this, and you'll be able to pass the "say the alphabet backwards" test if you ever get pulled over by the police for impaired driving!

In the cell below, write a loop to find the lowercase letters in our alphabet that have an ascii code that is divisible by 3. Useful tools:

- `ord()` – returns the ascii code of a character
- `chr()` – goes the other direction; returns the character of an ascii code
- `%` (modulo) – returns the remainder after division.

There's some partial code to get you started.

[98]:
```
ord('a')
ord('a')/3
```

[98]:
```
32.333333333333336
```

[94]:
```
chr(97)
```

[94]:
```
'a'
```

[118]:
```
alpha = 'abcdefghijklmnopqrstuvwxyz'
div_x_3_str = ''

# loop through the alphabet
for i in alpha :
    a_code = ord(i)          # convert letter to ascii code
    if a_code % 3 == 0:                         # test for divisibility (is that␣
 ↪a word?)
        div_x_3_str += chr(a_code)                      # add (or not) the␣
 ↪letter


print(div_x_3_str)
```

```
cfilorux
```

I can't tell if that's a new prescription drug or a cleaning product!

1.2.2 Repeating until a condition is `True` with `while`

a `while` loop runs until its test evaluates to false. Like this.

[39]:
```
test = 1

while test < 5 :
    print(f'test value = {test}')
    test += 1
```

```
test value = 1
test value = 2
test value = 3
test value = 4
```

Notice the increment operator `+=`.

`*variable* += *value*` is shorthand for `*variable* = *variable* + *value*`

A `while` loop is a bit dangerous because, if you mess up the test, it test may never evaluate to `False`, in which case the loop will run forever. This is an "infinite loop", and has been the bane of programmers ever since `while` loops were invented.

Useless trivia: The street address of Apple's headquarters in Cupertino CA is "One Infinite Loop".

We could use a while loop to keep prompting a user until they get something right.

[40]:
```
name = input('Please type your name.')
pswd = ''

while len(pswd) < 12 :
    pswd = input('Please enter your new password.')
    if len(pswd) < 12 :
```

```
        print('Sorry, the password must be at least 12 characters')

print(f'Hey, World! {name}\'s password is {pswd}!!!!')
```

```
Please type your name.Jay
Please enter your new password.234
Sorry, the password must be at least 12 characters
Please enter your new password.5tyf\
Sorry, the password must be at least 12 characters
Please enter your new password.rshghjvjyfygktdr
Hey, World! Jay's password is rshghjvjyfygktdr!!!
```

We can also add a `break` statement to exit the while loop immediately. The `break` command takes no arguments - when a `break` is encountered, the program exits the loop immediately.

In the cell below, see if you can modify our little password program above to exit if the user enters a 'q'. Make sure it also says something sensible when the user quits.

[48]:
```
name = input('Please type your name [q to quit].')

while name != 'q':
    name = input('Please type your name [q to quit].')
    if(name.lower() == 'q'):
        break
```

```
Please type your name [q to quit].q
```

1.3 Compartmentalizing and reusing code with `functions`

In any programming language, functions are ***great***. They are the greatest thing to be invented in programming since programming itself.

As you know, we use functions all the time. In this notebook, we've already used `input()`, `len()`, `print()`, `float()`, etc.

Functions wrap their code in a box so you don't have to worry about it; you just have to call the function and get/use/savor the result. Consider `len()` for example. How does it count the items? How does it know how to handle lists *and* strings? My theory involves elves but, the point is, `len()` is useful, and we don't need to dig into its underlying code unless we're curious.

The really great thing about functions is, as you also know, *we can write our own.* If you know that you are going to have do the same basic thing many times over, write a function! In fact, all of us that have been "in the biz" for a while have large collections of functions that we have developed over the years.

Let's just remind ourselves what a function looks like:

[49]:
```
def foo(the_arg) :
    print(f'The argument was {the_arg}')
```

We start with the `def` (**def**ine) keyword followed by the name of our function with any *arguments*

to the function enclosed in parentheses (the parentheses are required both in defining and calling the functions, even if their are no arguments).

Now we can call the function - just like it was any function in Python! Run the cells below to see our function in action!

[50]:
```
foo('Hello! I\'m a useless function!')
```

```
The argument was Hello! I'm a useless function!
```

[51]:
```
foo(22/7)
```

```
The argument was 3.142857142857143
```

Hm, looks like we overshot π by a little!

Since our little function only calls `print()`, it can handle anything print can handle.

[52]:
```
foo(['a', 'list'])
```

```
The argument was ['a', 'list']
```

Some functions, like `print()` and `foo()` are just verbs. They do something and that's it. Many functions, however, are useful in that they *return* an object. `len()`, for example, *returns* the length of the object passed to it as an argument.

We return a value using the `return` keyword.

Let's write a little function to compute the number needed to make the input number to it 11. Because everything should know how to go to 11!

[53]:
```
def goToEleven(notEleven) :
    x = 11 - notEleven
    return x
```

Now test our function in the cell below!

[56]:
```
goToEleven(9)
```

[56]: 2

Okay, let's play! Write a function `getpswd()` to get a password. It doesn't need any arguments; it should just be called and return the password after it's done its thing. So it should be called like this: `a_password = getpswd()`.

[85]:
```
def getpswd():
    a_password = input('password: ')
    print(a_password)
```

[86]:
```
# test your function here
getpswd()
```

```
password: 2ddddd
2ddddd
```