Chapter 10

Miscellaneous Topics II

In this chapter we will look at variety of useful things to know.

10.1 `str`, `int`, `float`, and `list`

The **`str`**, **`int`**, **`float`**, and **`list`** functions are used to convert one data type into another.

`str` Quite often we will want to convert a number to a string to take advantage of string methods to break the number apart. The built-in function `str` is used to convert things into strings. Here are some examples:

Statement	Result
`str(37)`	`'37'`
`str(3.14)`	`'3.14'`
`str([1,2,3])`	`'[1,2,3]'`

`int` and `float` The **`int`** function converts something into an integer. The **`float`** function converts something into a floating point number. Here are some examples.

Statement	Result
`int('37')`	37
`float('3.14')`	3.14
`int(3.14)`	3

To convert a float to an integer, the **`int`** function drops everything after the decimal point.

list The `list` function takes something that can be converted into a list and makes into a list. Here are two uses of it.

```
list(range(5))   [0,1,2,3,4]
list('abc')      ['a','b','c']
```

Examples

Example 1 Here is an example that finds all the palindromic numbers between 1 and 10000. A palindromic number is one that is the same backwards as forwards, like 1221 or 64546.

```
for i in range(1,10001):
    s = str(i)
    if s==s[::-1]:
        print(s)
```

We use the **str** function here to turn the integer `i` into a string so we can use slices to reverse it.

Example 2 Here is an example that tells a person born on January 1, 1991 how old they are in 2010.

```
birthday = 'January 1, 1991'
year = int(birthday[-4:])
print('You are', 2010-year, 'years old.')
```

The year is in the last four characters of `birthday`. We use **int** to convert those characters into an integer so we can do math with the year.

Example 3 Write a program that takes a number `num` and adds its digits. For instance, given the number 47, the program should return 11 (which is 4 + 7). Let us start with a 2-digit example.

```
digit = str(num)
answer = int(digit[0]) + int(digit[1])
```

The idea here is that we convert `num` to a string so that we can use indexing to get the two digits separately. We then convert each back to an integer using the `int` function. Here is a version that handles numbers with arbitrarily many digits:

```
digit = str(num)
answer = 0
for i in range(len(digits)):
    answer = answer + int(digit[i])
```

We can do the above program in a single line using a list comprehension.

```
answer = sum([int(c) for c in str(num)])
```

Example 4 To break a decimal number, `num`, up into its integer and fractional parts, we can do the following:

```
ipart = int(num)
dpart = num - int(num)
```

For example, if `num` is 12.345, then `ipart` is 12 and `dpart` is $12.345 - 12 = .345$.

Example 5 If we want to check to see if a number is prime, we can do so by checking to see if it has any divisors other than itself and 1. In Section 9.4 we saw code for this, and we had the following for loop:

```
for i in range(2,num):
```

This checks for divisibility by the integers 2, 3, ..., `num`-1. However, it turns out that you really only have to check the integers from 2 to the square root of the number. For instance, to check if 111 is prime, you only need to check if it is divisible by the integers 2 through 10, as $\sqrt{111} \approx 10.5$. We could then try the following for loop:

```
for i in range(2,num**.5):
```

However, this gives an error, because `num**.5` might not be an integer, and the `range` function needs integers. We can use `int` to correct this:

```
for i in range(2,int(num**.5)+1):
```

The `+1` at the end is needed due to the `range` function not including the last value.

10.2 Booleans

Boolean variables in Python are variables that can take on two values, `True` and `False`. Here are two examples of setting Boolean variables:

```
game_over = True
highlight_text = False
```

Booleans can help make your programs more readable. They are often used as flag variables or to indicate options. Booleans are often used as conditions in if statements and while loops:

```
if game_over:
    print('Bye!')
```

Note the following equivalences:

```
if game_over:          ⟺  if game_over==True:
while not game_over:   ⟺  while game_over==False:
```

note Conditional expressions evaluate to booleans and you can even assign them to variables. For instance, the following assigns `True` to `x` because `6==6` evaluates to `True`.

```
x = (6==6)
```

We have seen booleans before. The `isalpha` string method returns **`True`** if every character of the string is a letter and **`False`** otherwise.

10.3 Shortcuts

- **Shortcut operators** Operations like `count=count+1` occur so often that there is a shorthand for them. Here are a couple of examples:

Statement	Shorthand
`count=count+1`	`count+=1`
`total=total-5`	`total-=5`
`prod=prod*2`	`prod*=2`

  There are also shortcut operators `/=`, `%=`, `//=`, and `**=`.

- **An assignment shortcut**

  Look at the code below.

```
a = 0
b = 0
c = 0
```

  A nice shortcut is:

```
a = b = c = 0
```

- **Another assignment shortcut**

  Say we have a list `L` with three elements in it, and we want to assign those elements to variable names. We could do the following:

```
x = L[0]
y = L[1]
z = L[2]
```

  Instead, we can do this:

```
x,y,z = L
```

  Similarly, we can assign three variables at a time like below:

```
x,y,z = 1,2,3
```

  And, as we have seen once before, we can swap variables using this kind of assignment.

```
x,y,z = y,z,x
```

- **Shortcuts with conditions**

  Here are some handy shortcuts:

Statement	Shortcut
`if a==0 and b==0 and c==0:`	`if a==b==c==0:`
`if 1<a and a<b and b<5:`	`if 1<a<b<5:`

10.4 Short-circuiting

Say we are writing a program that searches a list of words for those whose fifth character is `'z'`. We might try the following:

```
for w in words:
    if w[4]=='z':
        print(w)
```

But with this, we will occasionally get a `string index out of range` error. The problem is that some words in the list might be less than five characters long. The following if statement, however, will work:

```
if len(w)>=5 and w[4]=='z':
```

It might seem like we would still get an error because we are still checking `w[4]`, but there is no error. The key to why this works is *short-circuiting*. Python starts by checking the first part of the condition, `len(w)>=5`. If that condition turns out to be false, then the whole `and` condition is guaranteed to be false, and so there is no point in even looking at the second condition. So Python doesn't bother with the second condition. You can rely on this behavior.

Short-circuiting also happens with `or` conditions. In this case, Python checks the first part of the `or` and if it is true, then the whole `or` is guaranteed to be true, and so Python will not bother checking the second part of the `or`.

10.5 Continuation

Sometimes you'll write a long line of code that would be more readable if it were split across two lines. To do this, use a backslash `\` character at the end of the line to indicate that the statement continues onto the next line. Here is an example:

```
if 'a' in string or 'b' in string or 'c' in string \
   or 'd' in string or 'e' in string:
```

Make sure there are no extra spaces after the backslash or you will get an error message.

If you are entering a list, dictionary, or the arguments of a function, the backslash can be left out:

```
L = ['Joe', 'Bob', 'Sue', 'Jimmy', 'Todd', 'Frank',
     'Mike', 'John', 'Amy', 'Edgar', 'Sam']
```

10.6 `pass`

The `pass` statement does nothing. Believe it or not, such a thing does have a few uses that we will see later.

10.7 String formatting

Suppose we are writing a program that calculates a 25% tip on a bill of $23.60. When we multiply, we get 5.9, but we would like to display the result as `$5.90`, not `$5.9`. Here is how to do it:

```
a = 23.60 * .25
print('The tip is {:.2f}'.format(a))
```

This uses the `format` method of strings. Here is another example:

```
bill = 23.60
tip = 23.60*.25
print('Tip: ${:.2f}, Total: ${:.2f}'.format(tip, bill+tip))
```

The way the `format` method works is we put a pair of curly braces `{}` anywhere that we want a formatted value. The arguments to the format function are the values we want formatted, with the first argument matching up with the first set of braces, the second argument with the second set of braces, etc. Inside each set of curly braces you can specify a formatting code to determine how the corresponding argument will be formatted.

Formatting integers To format integers, the formatting code is `{:d}`. Putting a number in front of the `d` allows us to right-justify integers. Here is an example:

```
print('{:3d}'.format(2))
print('{:3d}'.format(25))
print('{:3d}'.format(138))
```

```
  2
 25
138
```

The number 3 in these examples says that the value is allotted three spots. The value is placed as far right in those three spots as possible and the rest of the slots will be filled by spaces. This sort of thing is useful for nicely formatting tables.

To center integers instead of right-justifying, use the `^` character, and to left-justify, use the `<` character.

```
print('{:^5d}'.format(2))
print('{:^5d}'.format(222))
print('{:^5d}'.format(13834))
```

```
  2
 122
13834
```

Each of these allots five spaces for the integer and centers it within those five spaces.

Putting a comma into the formatting code will format the integer with commas. The example below prints `1,000,000`:

```
print('{:,d}'.format(1000000))
```

Formatting floats To format a floating point number, the formatting code is `{:f}`. To only display the number to two decimal places, use `{:.2f}`. The `2` can be changed to change the number of decimal places.

You can right-justify floats. For example, `{:8.2f}` will allot eight spots for its value—one of those is for the decimal point and two are for the part of the value after the decimal point. If the value is `6.42`, then only four spots are needed and the remaining spots are filled by spaces, causing the value to be right-justified.

The `^` and `<` characters center and left-justify floats.

Formatting strings To format strings, the formatting code is `{:s}`. Here is an example that centers some text:

```
print('{:^10s}'.format('Hi'))
print('{:^10s}'.format('there!'))
```

```
    Hi
  there!
```

To right-justify a string, use the `>` character:

```
print('{:>6s}'.format('Hi'))
print('{:>6s}'.format('There'))
```

```
    Hi
there!
```

There is a whole lot more that can be done with formatting. See the Python documentation [1].

10.8 Nested loops

You can put loops inside of other loops. A loop inside of another loop is said to be *nested*, and you can, more or less, nest loops as deeply as you want.

Example 1 Print a 10 × 10 multiplication table.

```
for i in range(1,11):
    for j in range(1,11):
        print('{:3d}'.format(i*j), end=' ')
    print()
```

A multiplication table is a two-dimensional object. To work with it, we use two for loops, one for the horizontal direction and one for the vertical direction. The print statement right justifies the products to make them look nice. The `end=''` allows us to print several things on each row. When we are done printing a row, we use `print()` to advance things to the next line.

Example 2 A common math problem is to find the solutions to a system of equations. Sometimes you want to find only the integer solutions, and this can be a little tricky mathematically. However, we can write a program that does a brute force search for solutions. Here we find all the integer solutions (x, y) to the system $2x + 3y = 4$, $x - y = 7$, where x and y are both between -50 and 50.

```
for x in range(-50,51):
    for y in range(-50,51):
        if 2*x+3*y==4 and x-y==7:
            print(x,y)
```

Example 3 A Pythagorean triple is a triple of numbers (x, y, z) such that $x^2 + y^2 = z^2$. For instance $(3, 4, 5)$ is a Pythagorean triple because $3^2 + 4^2 = 5^2$. Pythagorean triples correspond to triangles whose sides are all whole numbers (like a 3-4-5-triangle). Here is a program that finds all the Pythagorean triples (x, y, z) where x, y, and z are positive and less than 100.

```
for x in range(1,100):
    for y in range(1,100):
        for z in range(1,100):
            if x**2+y**2==z**2:
                print(x,y,z)
```

If you run the program, you'll notice that there are redundant solutions. For instance, $(3, 4, 5)$ and $(4, 3, 5)$ are both listed. To get rid of these redundancies, change the second loop so that it runs from `x` to 100. This way, when `x` is 4, for instance, the first value for `y` that will be searched is 4, rather than 1, and so we won't get the redundant $(4, 3, 5)$. Also change the third loop so that it runs from `y` to 100.

As you look through the solutions, you might also notice that there are many solutions that are multiples of others, like $(6, 8, 10)$, and $(9, 12, 15)$ are multiples of $(3, 4, 5)$. The following program finds only primitive Pythagorean triples, those that aren't multiples of another triple. The way it does this is every time a new triple is found, it checks to make sure that `x`, `y`, and `z` are not all divisible by the same number.

```
for x in range(1,100):
    for y in range(x,100):
        for z in range(y,100):
            if x**2+y**2==z**2:
                for i in range(2,x):
                    if x%i==0 and y%i==0 and z%i==0:
                        break
                else:
                    print((x,y,z), end=' ')
```

Example 4 In Section 15.7, we will write a game to play tic-tac-toe. The board is a 3×3 grid, and we will use nested for loops to create it.