Printing

Python offers several ways to print information. This handout presents three, using the `print` statement. First is unformatted printing, in which what you are printing is printed as is. Then comes formatted printing, which allows you to control how what you are printing looks. Finally, the `format` method provides the same capability. Python 3 programmers tend to prefer the latter when doing formatted printing, but either works in Python 3.

Unformatted Printing

Unformatted printing prints numbers and strings in a natural format: integers as ordinary numbers, floating point numbers with a decimal point, and strings as they are typed. Thus:

```
num = 1
flnum = 2.6
print("The integer is", num, "and the float is", flnum)
```

prints

```
The integer is 1 and the float is 2.6
```

A comma between two arguments prints a space. The `print` statement puts a newline after the output:

```
print("Here is one print statement")
print("And here is another")
```

prints

```
Here is one print statement
And here is another
```

Ending the `print` statement with a comma at the end of the line puts in a blank and suppresses the skip to the next line:

```
print("Here is one print statement ", end="")
print("And here is another")
```

prints

```
Here is one print statement And here is another
```

One problem with unformatted printing is that you cannot avoid adding spaces before numbers. So, if you try to print "$1.39" with this:

```
print("$", 1.39)
```

you get

```
$ 1.39
```

(notice the unwanted space). To print this, you need a formatted print statement.

Formatted Printing

A formatted print statement allows you to control how the number or string is printed. You can do lots of things with it that you cannot do with an unformatted print statement.

Let's start with the last unformatted print problem. We want to print "$1.39". Here's how we do it:

```
print("$%f" % (1.39))
```

This prints

```
$1.390000
```

The string immediately following the print is the *format string*. It is printed as typed, except when a "%" is seen. The sequence of characters following it control how the next arguments are to be printed. "%f" means to print a floating point number. "%d" would mean to print an integer as a decimal number ("d" stands for "decimal"). The "%" after the string says that a tuple containing the arguments for the printing follow.[1]

One problem with the above—too many decimal places. Let's restrict it to 2 decimal places by saying instead:

```
print("$%.2f" % (1.39))
```

This prints

```
$1.39
```

The "%f" says to print the number as a floating point number. The ".2" between the "%" and "f" means to print 2 digits after the decimal point. As another example, if you want to print the value of π to 7 decimal places, say:

```
import math
print("The value of pi is %.7f" % (math.pi))
```

You get

```
The value of pi is 3.1415927
```

Now suppose you want to print a table of numbers. You might say:

```
for i in range(1, 5):
        print("%.2f" % (math.pi * i))
```

You get this:[2]

```
3.14
6.28
9.42
12.57
```

That doesn't line up too well! We really want the numbers to the *left* of the decimal point to take up the same amount of room so the decimal points line up. So, we want to line the numbers up to the right. To do this, we first figure out how many characters (including the decimal point) we want the numbers to take up. That's 5 (because "12.57" takes 5 spaces to print and the rest take 4). So, we use the following format string:

```
"%5.2f"
```

When you make this change to the above `print`, you get:

```
 3.14
 6.28
 9.42
12.57
```

Now let's say we want to print several arguments. You do it this way:

```
print("%d + %d = %d" % (2, 2, 2 + 2))
```

gives

```
2 + 2 = 4
```

You can get very fancy:

```
print('%(language)s has %(Q)03d quote types.' % {"Q":2, 'language':"Python"})
```

[1] Actually, if there is only one argument to be printed, and that argument is a *number*, you can omit the parentheses. But if it's an expression, leaving the parentheses out could cause unexpected results, so it's better to put them in.

[2] Here's a good example of where the argument requires the parentheses even though only one number is being printed. Try leaving them out and see what happens.

gives

```
Python has 002 quote types.
```

The `'language':"Python"` associates the string "Python" with the name "language", so `%(language)s` means to print a string (the last "s") using the value associated with the name "language", which is the string "Python". Similarly, the name "Q" is associated with the integer 2, so `%(Q)03d` prints the value associated with "Q" (that is, 2) in a field of width 3 with leading 0s (the "03" in front of the "d").

Format Method

Python offers a second way to print. It uses a method called `format`.

Here's the basic form:

```
print( "{} is different than {}".format("Python 2", "Python 3"))
```

This replaces each "{}" with the argument to `format`, in the order of the arguments. So this prints

```
Python 2 is different than Python 3
```

If you want to refer to a specific argument of `format`, you can. Just put the number of the argument in the braces. So, in

```
print( "{0} {1} is different than {0} {2}".format("Python", 2, "3"))
```

the place holders {0}, {1}, and {2} are replaced by the first, second, and third arguments to `format`, respectively. Note that Python figures out the type of the data to be printed, and formats it accordingly. In the above, the "2" is treated as an integer, and the `"3"` as a string. Also, note you can print an argument multiple times, as is done with {0}.

Names work, too:

```
print('{language}s have {Q:03d} quote types.'.format(Q=2, language="Python"))
```

produces

```
Pythons have 002 quote types.
```

Note the syntax is different from what we saw earlier.

You can provide a format specification to control how the arguments to `format` are printed. For example:

```
import math
print("The value of pi is {0:.7f}".format(math.pi))
```

You get

```
The value of pi is 3.1415927
```

You can also control positioning within the field:

```
print("This prints to the left in the field:   '{0:<6}'".format('text'))
print("This prints to the right in the field:  '{0:>6}'".format('text'))
print("This prints in the center of the field: '{0:^6}'".format('text'))
```

produces

```
This prints to the left in the field:   'text  '
This prints to the right in the field:  '  text'
This prints in the center of the field: ' text '
```

Of course, this works with numbers too. It's particularly useful for printing tables!

What Is Actually Going On

Both of these are simply ways to write a string. Ignore the word `print`.

In the first method, the string in quotes is called the *format string* and the parenthesized list following the "%" are the *values*. The values are substituted into the format string, resulting in a new string. To see this, type the following to the IDLE interpreter:

```
import math
x = "2 * %f = %f" % (math.pi, 2*math.pi))
print(x)
```

and you will see

```
2 * 3.141593 = 6.283185
```

The same thing is true for the `format` method. Again, to see this, type the following to the IDLE interpreter:

```
import math
x = "2 * {0:.6f} = {1:.6f}".format(math.pi, 2*math.pi)
print(x)
```

and you will again see

```
2 * 3.141593 = 6.283185
```

Table of `print` Format Characters

Here is the list of the formatting characters and what they do.

character	*what it means*
b	Binary integer
c	Character
d	Decimal integer
e	Exponent notation, using 'e' to indicate exponent; default precision is 6
f	Fixed point notation; default precision is 6
n	Number; like d but it inserts the appropriate number separator characters
o	Octal integer
s	String
x	Hexadecimal integer; letters representing 10 to 15 are printed as lower case
X	Hexadecimal integer; letters representing 10 to 15 are printed as upper case