

CSC 241 Notes
Yosef Mendelsohn

With many thanks to Dr. Amber Settle for the original version of these notes.

Prof Settle's notes in turn are based on the Perkovic text.

# Functions

A mathematical function takes an input (say x) and produces some output (say x+1):

f(x) = x+5

To compute the value of the function for input 3, we write f(3) to obtain 8.

Functions in Python are similar.

We have already seen several functions in action such as **len()**, **print(), append()**, etc. We say that these functions are “built-in” or "predefined".

For example, the function **len**() takes a string or list as input and returns the length of the string or list.

The function **print**() takes a string prints it to the screen.

The function some\_list.**append**() takes an argument and appends that value to the list 'some\_list'.

Another built-in function is **min**() . This function that takes multiple inputs and returns the smallest.

See the example below:

>>> min(2,5)

2

>>> min(2, -4, 5)

-4

As mentioned, the above functions are “built-in” to Python. We sometimes call these “predefined” functions.

But what if we want to create our own functions?

Let’s take the example above that adds 5 to a number shown earlier:

f(x) = x+5

and create a function in Python that will do the same thing. Here is the code to do so:

def add5(x):

 return x+5

In general, a function that we define in Python begins with the keyword ‘def’.

A Python ‘def’ statement has the following format:

def <function name> (<0 or more parameters>):

 <iterated statements>

# one or more Python statements,

# all statements have a tab in front of them

**All functions that we create will adhere to the following pattern:**

* The **def keyword** is used to define a function.
* Following the def command is the **identifier of the function**. (In the case above the name of the function is ‘add5’).
* Following the identifier is a **set of parentheses containing 0, 1, or more** **input arguments**. (In the case above we have one argument, 'x').
* A **colon** is placed at the end of the first line. We sometimes call this line the function "header".

Below the def statement and indented with a tab is the **“implementation”** of the function, i.e. the Python code that does the work.

This indentation is absolutely required. Python notes the beginning and end of the function by looking at this indentation.

As we will see, Python uses indentation in many, many situations in order to determine the beginning and end of a section.

The **return statement** is optional. Some functions will need a return statement. Other functions will not. It depends on what the function is supposed to do. iIf you want the function to return a piece of information, then you will need a return statement. In the example above, we wish the function to return the value of x with 5 added on. Therefore, we will need to include a return statement.

**Examples**:

Write a function that takes 3 numbers as a parameter and returns the sum of those numbers:

def addThreeNumbers(n1, n2, n3):

 sum = n1+n2+n3

 return sum

**Exercise:**

Write a function that takes 3 numbers and returns (not outputs) the average of those numbers. Call the function average\_3\_nums.

**Exercise:**

Write a function that takes a list of numbers as a parameter and returns the average of the numbers in the list. Call it average\_of\_list.

Hints:

* If you have a list called 'my\_list', then you can get the sum of the list using Python's built-in function 'sum()' e.g. sum(my\_list)
* Recall that you can obtain the length of a list using the built-in function 'len()' e.g. len(my\_list)

**Thought Question:**

What would happen if the list was empty?

How do we fix this?

# Decision structures

Programs typically consist of a sequence of statements that the Python interpreter executes in order.

However, some alternative actions may need to be done based on some test. That is, we may have chunks of code that should be executed in certain situations, but should be skipped in other situations.

## One-way decisions

The ‘**if**’ statement is perhaps the most fundamental way that the Python interpreter can select which action to perform, based on a test.

**Example:** The following tests whether the input argument is less than 0. If it is, it prints a message that the argument is negative. Otherwise, nothing happens.

if n < 0:

print(‘n is negative’)

The **if statement** in its simplest form, has the following structure:

if <condition>:

<body>

<condition> is a Boolean expression, i.e. an expression that evaluates to True or False.

Some examples of such conditions include:

* if **n < 0**:
* if **age >= 86**:
* if **name == 'Smith'**:
* if **34 in some\_list**:

**<body>** consists of one or more statements that are executed if the condition evaluates to True. If the condition evaluates to False, then <body> will be skipped.

**Important:** As with functions, note how the statements inside the ‘if’ block must be indented.

if <condition>:

body statement 1

body statement 2

…

last body statement

rest of the program here

Draw a **flow chart** for the if statement.

Consider the following example program (**simpleTest.py**):

 n=42

 if n < 0:

 print("n is negative")

 print("These indented statements are part")

 print("of the if statement body")

 print("Still in the body...")

print("In the body no more")

print("I am executed whether or not the condition")

print("evaluates to True.")

Try sample runs on:

* Positive values (e.g. 4)
* Negative values (e.g. -4)
* n=0

The following is another example:

 secret="Djengo"

 guess = input("Enter the secret word: ")

 if guess==secret:

 print("You got it!!")

 print("Goodbye...")

**Exercise:** Modify this code to become a function that takes 0 parameters. Call it guess\_secret\_word()

**Exercise**: Modify the list function from our earlier exercise to check for an empty list and return 0.0 if the list is empty. Call the function average\_of\_list\_v2().

## A common mistake when writing functions

A common mistake for beginners writing functions is to use **print** instead of **return**

For example, if we define the function f above as:

def add5(x):

print(x+5)

then the function ‘add5’ will print x+5 on input x, but when used in an expression, it will not evaluate to it.

See the sample runs below to see what is meant by this:

>>> def add5(x):

 print(x+1)

>>> add5(3)

4

>>> 3\* add5(2)+1

3

Traceback (most recent call last):

 File "<pyshell#49>", line 1, in <module>

 3\* add5(2)+1

TypeError: unsupported operand type(s) for \*: 'int' and 'NoneType'

When evaluating add5(3), the Python interpreter evaluates add5(3) which will *print* 4 but does not evaluate to anything.

The second example makes this clear. The Python interpreter will print 3 when evaluating add5(2), but add5(2) does not evaluate to anything and so cannot be used in an expression.

**Exercise**: Does the print() function return anything?

**Answer**: The best way to check is to use an assignment statement so that the value returned by print gets a name:

>>> x = print("Hello!")

Hello!

>>> x

>>>

We can see above that x does not have a value, meaning that the print() function does not return anything.

# Iteration structures

Sometimes a program needs to repeat a block of code multiple times.

For example, the program may need to iterate through a list or some other collection of objects and run a block of code on the objects in the list.

A block code that repeats a number of times is typically called a **loop**.

As an example, consider the following:

>>> lst\_animals = ['cat', 'dog', 'chicken']

>>> for i in lst\_animals:

 print (animal)

cat

dog

chicken

Consider another example (**for.py**):

some\_list = [67, 34, 98, 63, 23]

for i in some\_list:

 print(i)

This code produces this output when run:

>>>

67

34

98

63

23

**An aside:**

You might note that the in the examples above, the identifier ‘i' is not very clear.

And… you’d be 100% correct. However, when iterating through a loop, programmers frequently use a simple identifier such as ‘i' or 'j' or 'x'. Sometimes it is fine to do this.

But in some cases, your code can be much more clear (and remember, we *really* like clear!) by using an identifier that reflects each item in the iteration. For example:

>>> lst\_animals = ['cat', 'dog', 'chicken']

>>> for item in lst\_animals:

 print (item)

I think the following is even better:

>>> lst\_animals = ['cat', 'dog', 'chicken']

>>> for animal in lst\_animals:

 print (animal)

Similarly, for our list of numbers, we might do something like:

some\_list = [67, 34, 98, 63, 23]

for num in some\_list:

 print(num)

It all depends on context – and very often, simple identifiers such as 'i' is just fine. But in general, having clear identifiers is quite nice.

If we’re iterating through say, a list of something where the values are not obvious (e.g. a list of meaningless numbers), then an identifier such as ‘i' is fine.

**Exercise**: Write a function that accepts a list of strings. The function will count all of the characters in every string in the list, and will return that value. For example:

list\_of\_strings = ["hello","how are you?", "all is good, thank you very much"]

print( count\_characters(list\_of\_strings) )

# will output **49**

As always, the solution can be found in this week's solutions file.

## The range function

Sometimes we want to iterate over a sequence of numbers in a certain range, rather than an explicit list.

For example, suppose we wanted to print all values between 0 and 9:

>>> for i in range(10):

 print(i, end=" ")

0 1 2 3 4 5 6 7 8 9

**Here is how the range() function works:**

1. If it is passed only 1 argument, it returns all values from 0 to the argument – non-inclusive. Eg: range(3) returns 0, 1, 2.
2. It it is passed 2 arguments, it returns all values from the first argument to the second argument (again, non-inclusive of the second argument). For example: range(1,5) returns 1,2,3,4.
3. It is is passed a 3rd argument, it returns from the first argument to the second argument (non-inclusive) jumping in increments of the 3rd argument. For example: range(5, 15, 3) returns 5, 8, 11, 14

Try the following using a for loop as demonstrated above:

1. From 0 to 1, i.e. 0, 1
2. From 3 to 12, i.e. 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
3. From 0 to 9, but with a step of 2, i.e. 0, 2, 4, 6, 8
4. From 0 to 24 with a step of 3, i.e. 0, 3, 6, 9, 12, 15, 18, 21
5. From 3 to 12 with a step of 5, i.e. 3, 8

As you will have noticed, the print() function prints the arguments on separate lines. A very convenient parameter to the print() function is **end=" "** says to end each line with a space. If we didn't do this, the default value for **end** is a new line. Try the code without that parameter and see what happens.

The answers are posted in this week's solutions under the function range\_practice1().