Chapter 2

Different types of values

By now you probably have a pretty good idea about what a value in Python is. So far you have seen text like `'Banana'`, integers like `7` and numbers with a fractional part like `4.25`.

In Python a text value is a type of value called a string which Python denotes `str` (abbreviation for “string”). So `'Banana'` is a string, and so is `'Banana split'`. There are two types of numbers in Python. Integers (like 7, 42, and 3) are called `int`. Numbers with a fractional part (like 3.1254 and 4.0) are that are called `float`, which is short for floating-point number.

We almost forgot `True` and `False`. As I mentioned earlier they are Python values too and are called booleans for no other good reason than being invented by an English mathematician called George Boole.

So the different types of values we know so far are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type in Python</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td><code>str</code></td>
<td>&quot;see you&quot;, '9'</td>
</tr>
<tr>
<td>Integer</td>
<td><code>int</code></td>
<td>0, 2721, 9</td>
</tr>
<tr>
<td>Floating-point</td>
<td><code>float</code></td>
<td>1.0, 4.4322</td>
</tr>
<tr>
<td>Booleans</td>
<td><code>bool</code></td>
<td>True, False</td>
</tr>
</tbody>
</table>

When you do computations in Python it is no problem to mix integers and floating-point numbers. Try this:

```python
print("What is 0.5 * 2?", 0.5 * 2)
print("What is 3 / 2?", 3 / 2)
```

As you can see we can also make computations using only integers that result in floating-point numbers. Some of the math operators not only work on numbers, they also work on strings. That way you can add two strings together. It is no longer math of course - but quite handy.

```python
fruit = 'Ba' + 'na' + 'na'
print(fruit)
```
Exercises

Exercise 2.1: What do you think is printed here?

```python
main_course = 'Duck a la Banana\n'
dessert = 'Banana split\n'
menu = main_course + dessert
print(menu)
```

Can you figure out what the special character `\n` represents?

Exercise 2.2: If you try to combine different types of values in ways that are not allowed in Python you will get an error. Try the following weird calculations and read the each error message carefully.

```python
x = 3 - '1.5'
print(x)

x = '23.3' - '4'
print(x)
```

Exercise 2.3: Write these two examples and compare the resulting values of `x`.

```python
x = '9' + '4'
print(x)

x = 9 + 4
print(x)
```

Exercise 2.4: Write these two examples. What happens in each case?

```python
x = '72' * 3
x = '72' * '3'
```

Exercise 2.5: Will this work? Use what you have learned from the other exercises and try to predict what will happen here. Then write the code and try it out.

```python
x = 'Ba' + 'na' * 2
print(x)
```
Having completed the exercises you should take note of the following four important points:

1. All Python values have a type. So far you know about strings, integers, floating-points, and booleans.
2. Math operators lets you do cool things like concatenating two strings by adding them together.
3. The flip side of that cool coin is that Python will assume you know what you are doing if you add two strings ("4" + "4" is '44' not 8) or multiply a string with an integer ('4' * 4 is '4444' not 16).
4. Python will throw a `TypeError` if you try to combine types values of values in ways that are not allowed. Remember this!

**FAQ - Frequently Asked Questions**

Q: If I have a variable, like `x`, how do I find out what type of value it represents?

A: You can use the builtin function `type`. Try `print(type(7))` or `print(type('Banana'))`.

**If-statement**

The small programs you have written so far all run the exact same sequence of statements (lines). Imagine if you could control which statements were run depending on the circumstances. Then you would be able to write more flexible and useful programs. Cue the music - and let me introduce the "if-statement".

Write the following carefully and see what happens. Notice the colon ending the if-statements. Also note that the lines below them are indented with four spaces. While you write, try to figure out what the if-statement does.

```python
bus_seats = 32
passengers = 20
bags = 20

print(passengers, "people ride the bus")

if bus_seats >= passengers:
    print('Everyone gets to sit down')

if bus_seats >= passengers + bags:
    print("Smiles, everyone has room for their bags beside them")

if bus_seats < 0.1 * passengers:
    print("Widespread dissatisfaction, some gnashing of teeth")
```

Try to change the values of `bus_seats`, `passengers` and `bags` and see what the program now prints.
You have probably realize that an if-statement creates a branch of statements that are only evaluated if the expression between the `if` keyword and the `:` reduces to `True`. If the expression between the `if` and `:` reduce to a value other than `True` or `False` then Python will interpret `0`, and `'` as False and all other non-zero and non-empty values as `True`.

**Exercises**

**Exercise 2.6:** Which of the following letters are printed: A, B, C, D, E, F, G. Make up your mind before you write and try the code.

```python
if 0:
    print('A')

if "Banana":
    print('B')

if 3.14159265359:
    print('C')

if False:
    print('D')

if 9 > 5 and 4 < 7:
    print('E')

if '':
    print('F')

if False or "banana":
    print('G')
```

**Exercise 2.7:** What happens if you do forget the `:` in the “if-statement”?

```python
if 4 > 2
    print('Hi!')
```

**Exercise 2.8:** What happens if you do not indent the code under the “if-statement”?

```python
if 4 > 2:
    print('Hi!')
```

**FAQ - Frequently Asked Questions**
Q: Isn’t "If" a poem by Rudyard Kipling?
A: Yes

Else-statement

Sometimes you not only want your program to do something if an expression reduces to True, you also want it to do something else if it is False. It is as simple as it looks:

```python
cookies = 3

if cookies > 0:
    print("Yum yum, I wonder if we have some milk too...")
else:
    print("WHO HAS TAKEN THE LAST COOKIES!?")
```

Remember to put a : after the else keyword. Write the code and change the value of cookies to 0 - I dare you!

Exercises

Exercise 2.9: Test your knowledge about expressions that reduce to a True or False value. Write the code below and then see how it responds to other values of x. Try to come up with other variations yourself.

```python
x = 0.0
# x = '0'
# x = ' '  # space
# x = ''   # empty string
# x = not 0
# x = 'zero'

if x:
    print('x is substituted with True in the if-statement')
else:
    print('x is substituted with False in the if-statement')
```

FAQ - Frequently Asked Questions

Q: Is “Else” a poem by Rudyard Kipling?
A: No

Functions
Now buckle down for the most powerful and useful thing in programming. Functions! They can be used in many ways. Here we will introduce you to their most common use. Functions serve as mini-programs that perform small well-defined tasks in your program.

I have started to write a song about functions:

```python
print("Functions are super, Functions are cool")
print("When writing a program they are a great tool")
print("La la dim du da da di")
print("Skubi dubi dumdi di")
print("Bing di dubi dum da di")

print("Functions are useful to package some code")
print("Although they seem strange your head won't explode")
print("La la dim du da da di")
print("Skubi dubi dumdi di")
print("Bing di dubi dum da di")
```

I am going to add many more verses and I do not want to have to write the entire chorus every time. So what would be more natural than to make a function called `chorus` that takes care of that for us. That way we can write our song the way lyrics with a chorus are usually written:

```python
def chorus():
    line1 = "La la dim du da da di"
    line2 = "Skubi dubi dumdi di"
    line3 = "Bing di dubi dum da di"
    return line1 + '
' + line2 + '
' + line3

print("Functions are super, Functions are cool")
print("When writing a program they are a great tool")
print( chorus() )

print("Functions are useful to wrap up some code")
print("Although they seem strange your head won't explode")
print( chorus() )
```

First, let us break down the function definition in the top part of this code:

1. We `define` a function with the `def` keyword (which is short for “define”, in case you wonder).
2. After `def` we write the name of the function. We call the function `chorus`, but we could name it something else like "refrain". A good name represents what the function does. Like good variable names, good function names can help you remember what your code does.
3. After the name you put two parentheses, `()`. 
4. Then a colon, `:`.
5. The statements that are nested under the `def` statement are indented with four spaces, exactly like we do under if-statements.

6. The return-statement ends the function and specify what the function produces when it is used. The expression after the `return` keyword reduces to a value that the function then returns.

When python runs this code, each line is executed one by one starting from the first line (remember oath two?). So in this case python first executes the definition of the `chorus` function. The only thing that has happened after Python has executed the first five lines is that it assigned the name `chorus` to the four indented statements. So Python now "knows" about the `chorus` function (like it "knows" about a variable `x` after we do `x = 4`).

To use the function, we "call" it by writing its name followed by parentheses: `chorus()`. When it comes to functions, saying "use", "call" and "run" means the same thing. As you can see, we call the function twice in the rest of the code. Each time we do, the following happens:

1. When a function is called each statement in the definition is executed one after the other. If you look at the function definition, you can see that our `chorus` function has four statements.
2. The first statement assigns a string value to the variable `line1`.
3. The second statement assigns a string value to the variable `line2`.
4. The third statement assigns a string value to the variable `line3`.
5. The fourth statement is a return-statement. The expression after the `return` keyword in the final statement reduces to a value and this value is what is returned back to where the function was called from.
6. So in our case this means that `chorus()` is substituted by a string with our chorus:

   "La la dim du da da di\nSkubi dubi dumdi di\nBing di dubi dum da di"

The key properties of functions are:

1. Functions name a piece code (some statements) just like variables name values like strings and numbers.
2. When a function is called it is substituted by the value that the function returns - the same way a variable in an expression is substituted by its value.
3. We call a function by writing the function name followed by parentheses: `chorus()`. Just writing the function name will not call the function.

**Exercises**

**Exercise 2.10:** Now that we have a `chorus` function we can use it many times in our code without having worry about how many "la la"s it has and so on. Try to change the "lyrics" in the chorus or add a new line to it. Notice how you only need to make changes in one place (the function) to change all the choruses in the song - cool right?
Try these changes to the chorus. For each example, go through the resulting code *slowly* repeating the breakdown above for each step in your code.

```python
def chorus():
    line1 = "La la dim du da da di"
    line2 = "Skubi dubi dumdi di"
    line3 = "Bing di dubi dum da di"
    line4 = "LA LA LA LA LA LA LA LA LA LA LA "
    return line1 + '\n' + line2 + '\n' + line3

def chorus():
    line1 = 'La la'
    line2 = 'Du bi du'
    chrous_text = line1 + '\n' + line2
    return chrous_text

def chorus():
    return "Hum hum\nMmm Mmm"
```

**Exercise 2.11:** What do you think happens if you move the definition of `chorus` to the bottom of your file. Make up your mind what you think will happen and why (maybe you remember what happens when you try to use a variable in an expression before you have defined it?). Then try it out.

```python
print("Functions are super, Functions are cool")
print("When writing a program they are a great tool")
print( chorus() )

print("Functions are useful to wrap up some code")
print("Although they seem strange your head won't explode")
print( chorus() )

def chorus():
    line1 = "La la dim du da da di"
    line2 = "Skubi dubi dumdi di"
    line3 = "Bing di dubi dum da di"
    return line1 + '\n' + line2 + '\n' + line3
```

Make sure you understand how the error you get relates to the way Python runs your script (remember oath two?)

**Exercise 2.12:** Try these examples of simple and - admittedly - not so useful functions, and see how each function call is subsituted by the value that the function returns.
```python
def meaning_of_life():
    return 42

answer = meaning_of_life()
print(answer)

def favourite_number():
    return 7
double_favourite = favourite_number() + favourite_number()
print(double_favourite)
```

**FAQ - Frequently Asked Questions**

**Q:** Can function names be anything?

**A:** Almost yes. Their names must be alphanumeric characters. I.e. letters, numbers and the \_ character. Good function names are lower case with \_ to separate words, like in the examples above. The same goes for variables.

**Functions can take arguments**

The functions we have written so far are not very flexible, because they return the same thing every time they are called. Now write and run this beauty:

```python
def square(number):
    squared_number = number**2
    return squared_number

print( squared(3) )
```

Notice how we put a variable ( `number` ) between parentheses in the function *definition*. This variable takes the value that we put between the parentheses ( `(3)` ) when we call the function. So when we call the function like this, then this implicitly happens: `number = 3`.

Here is another example:

```python
def divide(numerator, denominator):
    result = numerator / denominator
    return result

print(divide(44, 77))
```
So when we call this function, then these two things implicitly happen: \( \text{numerator} = 44 \) and \( \text{denominator} = 77 \).

A note on terminology:

- The variables in the definition line of a function, like \( \text{numerator} \) and \( \text{denominator} \), are called parameters.
- The values that we pass to the function in the function call, like \( 3, 44 \) and \( 77 \), are called arguments.
- It is crucial to remember that it is values not variables that are passed in and out of functions.
- You can define functions with any number of parameters as long as you use the same number of arguments when you call the function.

**Exercises**

**Exercise 2.13:** Try to call your `divide` function like this `divide(77, 44)`. What does it return and what do you learn from that?

**Exercise 2.14:** Try to call your `divide` function like this `divide(44)`. What happens what do you learn from that?

**Functions and variables**

Now you know about functions, how you pass values into them and how they return a value back. It is all a little mind-boggling, but you will get used to it once you do a lot of exercises.

Every time we call a function a new fresh mini-program springs into existence. This temporary little world only exists from the function is called (e.g. `meaning_of_life()`) and until it returns its value (e.g. \( 42 \)). This means that variables in your functions are not connected to the variables in rest of your script.

Running the following examples should help you understand this:

```python
def some_function():
    name = 'Dan'
    message = "{} my name is {}".format(greeting, name)
    return message

name = 'Kasper'
greeting = 'Hello'
print("Message:", some_function())
print(name)
```

Notice the following:
1. The message produced by the function greets Dan not Kasper.
2. From the last print-statement you can see that the \texttt{name} variable we defined in the first line of the script still has the value \texttt{‘Kasper’}, no matter what happens in the function.
3. Python could not find the \texttt{greeting} variable in the function but found it in the main script.

You should learn two rules from this example:

1. All variables that you assign values to in a function (including the function parameters) are \textit{private} to the function. If a variable in a function has the same name as a variable in the main script (like \texttt{name} above) then these are two \textit{separate} variables that just happen to have the same name.
2. When you use a variable in an expression like \texttt{greeting} and \texttt{name} in \texttt{"\{\} my name is \{\}".format(greeting, name)}, then Python looks for the variables inside the function. If it does not find a variable it will look in the main script too. In this case it finds \texttt{name} in the function and \texttt{greeting} in the main script. It is good practise to make your functions "self contained" so, in the sense that Python should not have to look outside the function for variables.

\section*{Chapter exercises}

The following exercises treat the areas we have worked on in this chapter. They are meant to train your familiarity with if-statements and functions. Remember that the purpose of the exercises is not to answer the questions but to train the chain of thought that allows you to answer them. Play around with the code for each example and see what happens if you change it a bit.

\textbf{Exercise 2.15:} What does this function do? How many parameters does it have? How many statements does the function have? What does the function print? Which value does it return? What is the difference between return and print?

\begin{python}
def power(a, b):
    print("This function computes {}**{}".format(a, b))
    return a**b

print(power(4, 2))
\end{python}

Try (possibly strange) variations of the code like the ones below to better understand the contribution of each line of code. You can begin with:

\begin{python}
def power(a, b):
    print "This function computes {}**{}".format(a, b)
    return a**b

print(power(4, 2))
\end{python}
and

def power(a, b):
    print "This function computes {}**{}".format(a, b)
    return a**b

result = power(4, 2)
print(result)

and

def power(a, b):
    "This function computes {}**{}".format(a, b)
    return a**b

print(power(4, 2))

and

def power(a, b):
    print "This function computes {}**{}".format(a, b)
    a**b

print(power(4, 2))

and

def power(a, b):
    return a**b
    print "This function computes {}**{}".format(a, b)

print(power(4, 2))

Exercise 2.16: Define a function called "diff" which takes two parameters, \( x \) and \( y \), and returns the difference between \( x \) and \( y \).

Example:
def diff(x, y):
    ...

diff(8, 2) # should return 6
diff(-1, 2) # should return -3
To test this, save the result of the function call in a variable and print it.

**Exercise 2.17:** Define a function called `all_equal` that takes five arguments and returns `True` if all five arguments have the same value and `False` otherwise. The function should work with any input, for example:

```python
call_equal("Dan", "Dan", "Dan", "Dan", "Dan")
call_equal(0, 0, 0, 0, 0)
call_equal(0.5, 0.5, 0.5, 0.5, 0.5)
call_equal(True, True, True, True, True)
```

Hint: You test equality with `a == b`. Now think back to what you learned about logic. Which operator can you use to ensure that `a == b` and `b == c`?

**Exercise 2.18:** Define a function called `is_even` which takes one argument and returns `True` if (and only if) this is an even number and `False` otherwise.

```python
is_even(8)  # should return True
is_even(3)  # should return False
```

**Exercise 2.19:** Define a function called `is_odd` which takes one argument and returns `True` if (and only if) the argument is an odd number and `False` otherwise.

```python
is_odd(8)   # should return False
is_odd(3)   # should return True
```

Can you use the function you just defined, `is_even`, to complete this exercise? How? Why is that a good idea?

**Exercise 2.20:** Define a function called `is_nucleotide_symbol` which takes one argument and returns `True` if this is either A, C, G, T, a, c, g or t, and `False` in any other case.

Name your parameters something sensible... e.g. `symbol`.

```python
is_nucleotide_symbol("A")  # should return True
is_nucleotide_symbol("B")  # should return False
is_nucleotide_symbol("Dan sucks")  # should return False
is_nucleotide_symbol(""")  # should return False
```

**Exercise 2.21:** Define a function called `is_complementary_base` which takes two parameters, `base1`, `base2`, and returns `True` if `base2` is the complementary of `base1`, and `False` otherwise.
is_complementary_base("A", "G") # should return False
is_complementary_base("A", "T") # should return True
is_complementary_base("T", "A") # should return False
is_complementary_base("Dan sucks", "A") # should return False

Can you use the function you defined in the previous exercise to complete this exercise? How? Why is that a good idea?

**Exercise 2.22:** Define a function called `is_series_cool`, which takes one parameter, `tv_series`, and returns `True` if (and only if) the parameter is either the string "Dr Who", "Doctor Who" or "Narcos".

is_series_cool("Days of our Lives") # should return False
is_series_cool("Doctor Who") # should return True
is_series_cool(42) # should return False

**Exercise 2.23** Consider this function definition that takes a single number as argument:

```python
def square(n):
    return n**2
```

What does it do? What does it return? What number does `square(2)` then represent?

Below I have used it in some expressions that are printed. Make sure you understand what each expression evaluates to. Do the explicit substitutions and replacements on paper before you run it. Remember that we can substitute a function call (like `square(2)`) for the value it returns, just like we can substitute a variable `x` for the value it points to.

```python
print(square(3))
print(square(2 + 1))
print(square(2) * 2 + square(3))
print(square(square(2)))
print(square(2 * square(1) + 2))
```

**Exercise 2.24:** Make a function called `teaching_role` that takes a string argument that is either "Kasper", "Dan" or "Asger". If the argument is "Kasper" the function should return "lecturer", otherwise it should return "TA". If you do that then the code below should print `Dan is a TA`. You will need if-statements and else-statements to write this function. Try changeing the name to Kasper and Asger to test if your function works.
name = 'Dan'
print("{} is a {}".format(name, teaching_role(name)))

Exercise 2.25: Define a function called `celcius2fahrenheit` that converts from celcius to fahrenheit. You can do this because you know the linear relationship between the two. One the figure below you can see that the slope is $\frac{5}{9}$ and the intercept is 32. The function should have one parameter `celcius`. Inside the function you should define the variables `slope` and `intercept` and give them the appropriate values. then you can do the calculate the conversion to Celcius using these variables and `return` the result.

![Graph showing linear relationship between Celcius and Fahrenheit]

Exercise 2.26: Try to change your conversion function so it takes three arguments, corresponding to `celcius`, `slope` and `intercept` so you can call it like this to convert 27 degrees celcius: `conversion(37, 5 / 9, 32)`. Now you have a function that can do any linear conversion that you can put inside an other function like this:

```python
def celcius2fahrenheit(celcius):
    return conversion(celcius, 5 / 9, 32)
```

Exercise 2.27: Now try to extend this to a different problem: It has been found that the height and weight of a person are related by a linear equation with slope = 0.55 and intercept = -25. Define a function called `predict_weight` which takes just one argument, the height of a person, and returns the estimated weight of the person.

```python
def predict_weight(height):
    # Your implementation here
```