# Basic data manipulation in Python

## Basic numerical types

There are many ways to store data in Python, or any programming language. The most simple is to assign a variable to a single number, `the_number = 87` . There are two general classes of numbers recognized by computers; integers and floating point numbers. An integer is exactly what it sounds like, but they come with some caveats. In Python 2 `3/2` is equal to `1` . This seems a bit nonsensical unless you're familiar with the quotient-remainder theorem. `2` goes into `3` once, with a remainder. In Python 3, all use of the `/` is assumed to be floating point division, so `3/2` will behave normally. If you want to actually do integer division use `//` , and use `%` to find the remainder. The `%` is useful for checking if an integer is a multiple of another.

Floating point numbers are sort of like decimals. The only caveat is that because we have finite memory and computers work in binary, floating point numbers are stored in base 2 and rounded after every operation. Normally this provides way more precision than you will ever achieve because of error in the data, but it can produce some slightly surprising though unimportant effects.

We can of course do operations on these basic numbers, and all the logic extends (as it always will) the the variable names that they are assigned to.

```
test_variable = 2
other_thing = 14.5e10
print(test_variable + other_thing)
print(test_variable / other_thing)
```

Python supports all the basic mathematical operations, `+` , `-` , `*` , `/` , and exponentiation with `**` . Square roots can be computed with `sqrt(some_number)` , or with an exponent of one-half. Python follows standard order of operations, but feel free to be generous with your parentheses especially where they make reading your equations easier. Also use spaces around operators where it improves your ability to read code; it doesn't affect what the computer actually does but it does affect your ability to fix mistakes and make tweaks. Python also provides another set of operators for writing neater code, `+=` , `-=` , `*=` , and `/=` . These statements are exactly equivalent and the logic applies for the rest:

```
variable1 = variable1 + variable2
variable1 += variable2
```

## Boolean types

There's another type of variable we'll use often, which is the boolean, which can only be `True` or `False` . Booleans can just be created with something like `is_apocalypse = False` (I hope). Beware though! `True` and `False` are not reserved words, so you are not prevented from doing something like `True = False` , then later assigning some `test_flag = True` . In this case `test_flag` is set to the boolen value `False` . Boolean values are typically generated by using the binary comparison operators (binary because they only compare two things), `==` , `>` , `<` , `>=` , `<=` Most boolean variables or expressions can be combine with `and` and `or` , but numpy ndarrays as we will see later need `&` and `|` , (called bitwise versions of `and` and `or` ) because `and` / `or` try to convert the two things being compared to a single boolean value before comparing them, and it's ambiguous how to convert an entire array to a single truth value.

## Native Python data structures

### Strings

I'm going to be using strings a lot so I might as well say something about them here. For our purposes, strings are a way to store text in a variable. They're so much more than that, but that will suffice for now. Strings are any characters surrounded by single `'` or double quotes `"` . Strings can be indexed, so if I have some `test_string = 'test'` , I can ask for `test_string[0]` and I'll get back another string, `'t'` . Strings are immutable, but can be concatenated. This is done using the `+` operator. You could say that "adding" strings is concatenating them, while "adding" integers/floats is numerical addition. At some point in the future you will want to put variables in strings. This can be done in a multitude of ways, from using the `%` as in C, using f-strings and literal interpolation.

## Tuples

The simplest Python data structure is the tuple. Any variables just separated by a comma, and optionally surrounded by parentheses are a tuple. For example, `my_tuple = 1,2,3,4` . A tuple is ordered, and so can be indexed. To get the first entry in this tuple, use `my_tuple[0]` , which in this case will be `1` . If we want to know how long the tuple is, Python provides `len()` , which you can give a tuple to ask for its size. In this case `len(my_tuple)` would be 4. An empty tuple can be created with `(,)` .

## Lists

A step up from tuples is the list. A list looks a bit like a tuple, but its values must be surrounded by hard/square brackets. For example, `my_list = [1,2,3,4]` . Now that we know of two data structures, we can make use of the Python functions that convert other structures. `list(my_tuple)` returns a list idential to `my_list` , and `tuple(my_list)` returns a tuple identical to `my_tuple` . Lists work just like tuples but are mutable, so we can index them and also assign values. So `my_list[0] = 0` will change first value of `my_list` to `0` . The same `len` function works on lists, but for lists we can also change the length by appending and extending. So if we do `my_list.append(20)` , now `my_list` is `[0,2,3,4,10]` . Lists can also be combined with extend, so `my_list.extend(my_list)` makes `my_list == [0,1,2,3,4,0,1,2,3,4]` . An empty list can be created with `[]` .

## Dictionaries

Dictionaries look like magic compared to tuples and lists because they've hidden a lot of complicated math behind a simple interface. I won't get into how they work here, but for those who know: they're associative arrays and use a hash. Basically, a dictionary is a bunch of key, value pairs. The keys are what are put in the square brackets, and the values are what you get back from the dictionary. While tuples and lists are indexed with and only with integers, dictionaries are indexed by anything in their keys. Therefore, a dictionary's key and value must be assigned at the same time and thus they do not support things like appending and extending (in such a simple fashion as lists). The simplest way to use a dictionary is to create it then assign keys and values using the square brackets.

```
num_images = dict()
# OR
num_images = {}
num_images['HD_10700'] = 13
num_images['HD_172167'] = 49
```

Now we can ask for `num_images['HD_10700']` and we'll get back `13` . Dictionaries can also give you lists of their keys and values, using `num_images.keys()` and `num_images.values()` .

## Sets

Tuples and lists hopefully make a lot of sense, dictionaries are a bit esoteric, and sets are even stranger. Python's sets are just like those in math; they're unordered and contain no duplicates. Sets also come with a whole host of operations modeled directly after those in math; union, intersection, subset checking, etc. Sets are very useful when you only care about checking if some value is another group, since that operation is faster with sets than lists and tuples. Since sets are unordered, they can't be indexed, and so things must be either in a set when it is created or added to an existing set with `my_set.add(some_element)` .

## defaultdict

The defaultdict is part of Python's `collections` module, so use the import statement `from collections import defaultdict` to use it. The value of `defaultdict` is that you can set a default value for those that don't exist yet. I have used defaultdict to keep track of files, for example where each of the keys in the dictionary is a filter or star, and the value of the key is a list of file names. You can then always do `my_defaultdict[target_name].append(new_filename)` , even if you've never assigned `my_defaultdict[target_name]` before.