Introduction to Data Types - April 18, 2020

Hi everyone! As you are going through the Python courses on edX, you have already been working with some of the most useful **data types** in Python, including *integers*, *floats*, and *strings*. Data types are a core component of any programming language, allowing you to store and manipulate many different kinds of information, like numbers, words, and lists. Here, we'll discuss how data types are implemented in Python and provide a reference for some of the most useful data types for biological data analysis.

How do data types work in Python?

At first, the existence of different data types in Python may not be obvious. In many other programming languages, such as C++, data types are *explicitly* defined when you declare a variable. For example, if we want to make a new variable `num` to hold the integer number 5, we would write

```
int num = 5;  // c++
```

In C++, we have to tell the compiler what type the variable is before we can use it, and we do that here with the `int` keyword. The developers of Python saw this code structure as redundant: we already know that `num` is an integer because we are giving it the value of 5. Thus, when you are declaring a variable in Python, the type identifier is omitted for simplicity:

```
num = 5
```

Also, since Python is a scripting language, it can do something called **dynamic memory allocation**: rather than having to block off memory in the computer before the program runs, Python can grab the memory it needs as it progresses through your script. Different data types require different amounts of space in memory; however, since Python allocates memory dynamically, we are free to change the type of a variable whenever we want. For example,

```
my_var = 5  # my_var is declared as an integer
my_var += 2  # we can add 2 to my_var to make it 7
my_var = "blue"  # now it's a string!
```

Just because data types are *implicit* in Python doesn't mean they aren't important. For example, if you try to run something like this,

```
my_int = 5
my_string = "blue"
print(my_int + my_string)
```

you'll get the following error:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-adf924bcdbe7> in <module>()
----> 1 print(my_int + my_string)
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

If we try to add a string to an integer, Python doesn't know what to do! Data types define not only the kind of information contained in a variable, but also how you use it, the methods available to it, and how it interacts with other variables. Mastering the most common types will allow you to work efficiently with any kind of data that you will come across in your projects.

Data types reference sheet

Different data types have different use cases, and some are more efficient than others for specific operations. In addition, while Python itself has many data types that are broadly useful, external libraries also can provide their own data types that add extra functionality and efficiency for specific types of analysis. In this section, we'll provide a reference for some of the most common data types in computational biology, and we'll go over when they are best to use.

Python built-in types

- **Boolean (`bool`):** either `True` or `False` . This is the simplest data type, with only two possible values. Booleans are the foundation of programming logic: for example, booleans allow the computer to decide if code in an `if` block should be run, or if a `while` loop should end. Any comparison we perform (such as `my_int < 5`) will give either `True` or `False` , allowing us to run some code only when a condition is met, or run other code when it is not.
    - Example declaration: `my_bool = True`
    - Booleans are also useful as *flags* that we can set in response to something in our data to be retrieved later. For example, if we want to include a line in our output file about *TARDBP* read counts only if our data has at least one read that maps to that gene, we might make a boolean variable called `data_has_TARDBP` to keep track of this. We could initialize it as `data_has_TARDBP = False` and then flip it to `True` if we detect a *TARDBP* read at any point in our data analysis. At the end of the script, we would then check if this flag is `True` using `if` , and modify our output accordingly.
    - To check if a boolean variable (eg. `my_bool`) is True, simply write `if my_bool:` . To check if it's False, use the `not` keyword (`if not my_bool:`).

- **Integer (`int`):** any whole number, including positive and negative numbers (eg. 5, 0, -107). Integers are commonly used for counting, indexing iterable types like lists (eg. `my_list[1]`), and iterating through indices in a `for` loop using `range()` . Unlike most programming languages, Python does not place a limit on the largest possible integer.
    - Example declaration: `my_int = 2`

- **Floating-point (`float`):** a decimal number (eg. 1.1, 3.14159265, 2.998e8). Floats are the basic data type for storing decimal numeric data, like physical measurements, ratios, and distances. Floats in Python store 53 bits of information, or approximately 16 significant digits of precision.
    - Example declaration: `my_float = 1.1`
    - In Python 3, fractions of integers automatically evaluate to floats (eg. 3/7 becomes 0.42857142857142855).
    - In order to create very large or very small numbers, you can also use scientific notation to declare floats. To do this, place an `e` in between the decimal number and the exponent (eg. `5.4e10`).

    - **Warning**: Computers fundamentally store information using binary representation (0's and 1's), not in the base 10 that we humans work with, and this can cause some funny behavior if we don't structure our code carefully. For example, if we add the float `0.001` to a variable `var` 1000 times using a `for` loop, the variable's value after the loop completes is `1.0000000000000007`, rather than exactly 1. This is because the binary representation of the number `0.001` that the computer creates is only a high-precision approximation, and the error quickly becomes visible after only a small number of mathematical operations. In our code, if our goal is to check if `var` is equal to 1, we probably won't get the outcome we expect. Instead, we should check if `var` is *extremely close* to 1, using something like `if abs(var - 1.0) < 1E-8:`. This type of comparison is also implemented in external libraries, including `math` (eg. `math.isclose()`).

- **String (`str`):** an ordered collection of characters (eg. `"this is a string"`). Strings are the fundamental type for storing text data, including letters, words, phrases, and sentences, as well as sequencing reads, sequence alignment information (eg. CIGAR strings), and file paths.

    - Example declaration: `my_string = "blue"`
    - In Python, strings can be declared using either single quotes (`'blue'`) or double quotes (`"blue"`). For example, if we want to store the sentence "It's a wonderful life!" as a string, we can enclose the string in double quotes to avoid problems with the apostrophe.
    - If we need to use a character in a string that messes with the string declaration (eg. both single and double quotes in a single string), we can *escape* the problematic character by preceding it with a backslash `\`.
    - Basically any character we can type on a keyboard, as well as many other special characters (eg. symbols, tabs, newlines, Unicode), can exist in a string. Newlines (a "character" that tells the computer to start writing text on the next line) are represented by `\n`, and tabs are represented by `\t`.
    - Characters in a string can be retrieved with an index (eg. `my_string[0]` returns the first character in the string).
    - Substrings can be retrieved by *slicing* (eg. `my_string[0:3]` returns the first three characters in the string).
    - Strings can be *concatenated* by adding them (eg. `'a' + 'b'` returns `'ab'`).
    - In Python, strings are *immutable* objects, meaning that they cannot be changed once they are created. This means that if we try to change the value of a character in a string by indexing (eg. `my_string[0] = "A"`), Python will throw an error. We can often get around this, however, by redefining an existing string variable with new data. For example, to change the first letter of `my_string` to A, we can write `my_string = "A" + mystring[1:]`.

- **List (`list`):** an ordered collection of items of any type (eg. `[0, 1, 2, 3]`, `[0, 1.0, "red", True]`). Lists are a very convenient data type to store ordered sequences, such as the lines in a text file, sequencing reads, gene names, measurements, read counts, RGB color values—pretty much anything you can think of! Items in a list don't even have to be the same type. Lists are the fundamental *iterable* type in Python, meaning we can run through its items with a `for` loop and perform some operations on each item in sequence. However, with this convenience comes relative inefficiency in both memory usage and time: for processing large and/or multidimensional data, like 3D microscopy images or large sequencing datasets, lists often don't make the cut.

    - Example declaration: `my_list = [0, 1, 2, 3]` (enclose the list in *square brackets*, and separate items with a comma)
    - Lists can be *indexed*, *sliced*, and *concatenated* just like strings, with exactly the same syntax.
    - Additionally, unlike strings, lists are *mutable*, meaning we can change the items whenever we want. For example, we can change the first element of `my_list` to 5 by indexing (eg. `my_list[0] = 5`).
    - We can also change the length of the list at any time. We can add an element to the end using the `.append()` method (eg. `my_list.append("a new value")`, insert an element at any position using `.insert()`, and remove an element using `.pop()` or `.remove()`.

    - One common method to produce a list is to start with an empty list and append items into it one-by-one. For example,

      ```
      odds = []
      for i in range(10):
          odds.append(2*i+1)  # [1, 3, 5, 7, ...]
      ```

      The developers of Python saw an opportunity to make this code simpler, so they created a syntax called **list comprehension**. This structure allows you to create and populate a new list in a single line of code by embedding the `for` loop inside the list declaration, like this:

      ```
      odds = [2*i+1 for i in range(10)]  # also [1, 3, 5, 7, ...]
      ```

      This makes the code both more concise and easier to read (at least once you've seen it enough times!). List comprehensions can get even more powerful: you can even work in `if` conditionals and multiple `for` loops. If you want to learn more about this, check out this resource.

- **Tuple (`tuple`):** an *immutable* ordered collection of items. Tuples function similarly to lists, except once declared, the items of a tuple can't be changed. Many functions return tuples in order to return multiple values at the same time: for example, the curve-fitting function `scipy.optimize.curve_fit()` returns the tuple `(popt, pcov)` to provide both the fitting parameters and their covariance matrix.

    - Example declaration: `my_tup = (1, 2, 3)` (enclose the list in *parentheses*, and separate items with a comma)
        - In many cases, the parentheses are actually optional. For example, when using the iterator `enumerate()` to iterate over a list's indices and items together, we often write this as

          ```
          for i, item in enumerate(my_list):
              # do some stuff with index 'i' and item
          ```

          Here, `i, item` is actually a tuple, and the parentheses are omitted.

- **Set (`set`):** an *unordered* collection of unique items. Sets maintain information about which elements are in the set, but not where they are relative to each other. Additionally, sets can only hold one of each object (eg. there cannot be two 7s in a set). Sets are much more time-efficient than lists when all we need to know is whether an element is part of the set or not. Sets also enable set arithmetic, such as unions and intersections.

    - Example declaration: `my_set = set([1, 2, 3])`
        - Here, we see that Python has not implemented a syntax for creating sets directly. Rather, sets are created by using the **constructor** `set()` on another iterable type, like a list or a tuple.
    - We can take advantage of the uniqueness requirement of sets to remove duplicate items from lists. For example, consider a list `my_list` that contains the numbers `[0, 0, 1, 1, 2]`. If we want to get a new list that contains all the unique elements of `my_list`, we can cast the list to a set, and then cast it back to a list:

      ```
      unique_list = list(set(my_list))  # this list now contains 0, 1, and 2 only
      ```

      However, because sets are an unordered type, this will scramble the order of the items. If we need to, we can use the `sorted()` function to put them back in order.

- **Dictionary (`dict`):** an unordered collection of items accessed using keys. Like lists, dictionaries can hold any number of values of any type. However, rather than being indexed using integers, dictionaries are indexed by *keys*, which can be any immutable type (eg. integers, floats, strings, and tuples). Strings are often used as keys. For example, if I wanted to store information about the colors of all the objects in my kitchen, I could make a dictionary called `kitchen_colors`:

```
kitchen_colors = {'spatula':'black', 'sink':'silver', 'refrigerator':'grey', ... }
```

If I wanted to get the color of my spatula, I would look it up in the dictionary using the appropriate key:

```
color = kitchen_colors['spatula']  # 'black'
```

Dictionaries are a highly flexible way to store heterogenous data for quick access. However, they do not store any information about order.

  - Example declaration: `my_dict = {"fruit":"strawberry", "color":"red", "flavor":"sweet", "count":56}` (enclose the dictionary in *curly brackets*, with keys and values separated by a colon and pairs separated by commas)
  - Dictionaries are organized as key-value pairs, where the key is used to look up the value. We can think of this as similar to a real-world dictionary, where the word is the key and the definition is the value.
  - In Python, dictionary values can be any type, including lists and dictionaries!
  - If we want to iterate over a dictionary, we can do it in multiple ways. We can iterate over its keys using the `.keys()` method, its values using `.values()`, or its key-value pairs (as a tuple) using `.items()`. Keep in mind that these aren't going to be in any particular order.

- **NoneType (`NoneType`):** the type of Python's null object `None`. We don't really need to worry much about this, but we do need to be aware that it exists. `None` is the object Python gives variables that have *no value*. This can come up in some corner cases: for example, if our code relies on user input and the user fails to provide any input, our input-catching variable may end up with the value `None`. Also, functions that do not return anything (eg. in-place methods like `list.append()` or `list.sort()`) automatically return `None`.

  - We can't make any variables with the type `NoneType`. This type is reserved for `None` only.
  - We can check if a variable `var` is `None` using `if var is None`.

External data types

While Python provides many broadly useful types (including some not mentioned above), certain analyses require specially designed data types that provide additional functionality or streamline efficiency of data processing. Here, we describe just a few of the many data types provided in external Python packages.

- **NumPy array (`numpy.ndarray`):** an ordered *n*-dimensional array of values with the same type. Requires `numpy`. NumPy arrays have a couple more restrictions on their use than lists (eg. all items in the array must have the same type); however, these restrictions greatly boost their efficiency, allowing us to process very large amounts of data quickly and with less memory usage. Additionally, arrays can be multidimensional (eg. 2D, 3D, …), making them the perfect data type to represent microscopy images. Together with the rest of the `numpy` and `scipy` packages, arrays enable fast matrix operations, including matrix multiplication, filtering, thresholding, deconvolution, etc. Check out the documentation if you want to learn more.

- **pandas DataFrame (`pandas.DataFrame`):** a 2-dimensional table of heterogeneous data. Requires `pandas`. DataFrames act like spreadsheets, with columns of data labeled by headers. We can perform operations to generate new rows and columns, similar to what we might do using Excel. DataFrames are a convenient way to organize large tables for manipulating data, plotting, and outputting to files. Here is a tutorial on how to get started using this data structure.

- **BioPython Seq (`Bio.Seq`):** a more advanced string-like object for working with biological sequence data. Requires `biopython`. In addition to the sequence itself, Seq also allows us to set a biological alphabet (eg. DNA, RNA, or protein), which can help us catch errors in our code or data. Seq also provides additional methods that we might find useful, like `.reverse_complement()`, `.transcribe()`, and `.translate()`. Information about the Seq object, as well as many other useful constructions for sequence analysis, can be found in the BioPython documentation.