Day #4: Variables, Operators, and More

Corey Predella & Gunner Peterson

Abstract Academy

ΔΣ Introduction

Variables within the python programming language will be the foundation behind all of your code. In a nutshell, variables allow you to **store, reference, and change data,** which is the very basis of all programming. Since computer programs are all about automating solutions to problems, variables will be critical to your creations.

ΔΣ Declaring Python Variables

```
age = 16
```

Listing 1: Declaring a Python Variable

In python, variables are always initialized with a three-step process. Notice above the variable says, "age = 16" The first part of this initialization is naming the variable. In this case, we named the variable, "age". Now, whenever you call this variable, it will reference the value assigned to it. The next step is to actually set our variable equal to something. We use the, "=" operator to do so. The final step is to choose a piece of data to assign our variable to. In this case, we chose the integer 16, however, data doesn't have to be restricted only to integers, but more on this later. Here are some examples of creating variables:

```
name = "Corey"    # example of a string
age = 16    # example of an integer
has_pets = True    # example of a boolean
height = 6.2    # example of a floating point integer
sadness = null    # example of a null value
```

Listing 2: Creating Different Variables

To output your variable to the console, use the print() method. This method takes in an input, which is the data you are going to print. You can pass through raw data or variable names. In this case, we want to print our variable, "age", so we say

```
print(age)    # output will say "16"
```

Listing 3: Using the Print Statement

ΔΣ Datatypes

A datatype is a format of data. Inherently, a computer recognizes integers and strings as different objects because they require different amounts of bits and bytes.

Definition 1 (*Bits and Bytes*)

A bit is a binary 0 or 1 value, and a byte is a sequence of 8 bits. Bits are often used to represent base-10 numbers in binary, because computers perform equations in base-2.

I highly **recommend** you read about number bases so you can understand the following results with binary numbers.

Theorem 1 (*Base-2 Representation*) Let $n_k, n_{k-1}, ..., n_1$ be a base-2 number such that $n_k, n_{k-1}, ..., n_1 \in \{0, 1\}$. Then,

$$n_k n_{k-1} n_{k-2} ... n_1 n_0 = 2^{k-1} n_k + 2^{k-2} n_{k-1} + ... + 2^1 n_2 + n_1$$

in base-10.

Result 1 $00101101 = 2^7(0) + 2^6(0) + 2^5(1) + 2^4(0) + 2^3(1) + 2^2(1) + 2(0) + 1 = 45$

Now you know that the number 45 requires exactly 6 bits. However, computers store data only in bytes in powers of 2, so 45 is stored as an 8-bit integer. There are 8-bit integers (1-byte integers), 16-bit integers (2-byte integers), 32-bit integers (4-byte integers), and so on and so forth.

Problem 1 (*Check Your Understanding*) Why is 65537 an annoying number for computers?

Solution 1 (*Check your Understanding*) $65537 = 2^{16} + 1$ and $2^{16} + 1 \equiv 1 (\mod 2^8) \implies$ 65537 is stored in 32 bits instead of 16 bits.

Essentially, all forms of data are represented in binary, and the problem of data storage is the question of "how can we most optimally store information?" In 1992, the JPEG file format revolutionized how image data was stored. People realized that with some knowledge and intelligence, data can be stored in brilliant ways. Most recently, MIT has found that converting binary data into genetic DNA can allow for extremely compact storage. They claim that all of the computational data in the world can be stored into a coffee cup of DNA[1]. The essential data types of Python are:

- **Integers:** Numbers that do not have a decimal
- **Floats:** Numbers that do have a decimal
- **Strings:** Any sequence of ASCII characters encased by double or single quotes
- **Boleans:** True or False values. In computer science, non-zero numbers are True and zero value numbers are False. Python recognizes "True" and "False" as the respective keywords for true and false values

Refer to the second code listing for examples of these data types in action. In reality, there are many more data types. For our purposes, we will focus on these four types.

ΔΣ Choosing the right datatype

When programming, it's important to think about what data types you should use. For instance, let's consider your current emotion as a piece of data. Which type of data would suffice to represent your emotion? If you thought to represent your emotion as a string, you are correct. Here are some other ways to represent emotion as data:

```
1 emotion = "curious"   # this way of representing emotion is a good idea if you want to be
      specific
2 happiness = 5.67 / 10   # this represents your happiness specifically within the context of
      a range from 0-1
3 happy = True   # this represents your specific emotion as a binary value; either you are
      happy, or you aren't
```

Listing 4: Representing Information as Data

ΔΣ Flexibility of Python Variables

Because Python is a high-level programming language, you can usually get a lot done in one line. For instance, consider the following program:

[1] Banal, J.L., Shepherd, T.R., Berleant, J. et al. Random access DNA memory using Boolean search in an archival file storage system. Nat. Mater. 20, 1272–1280 (2021). https://doi.org/10.1038/s41563-021-01021-3

```
var = 10 + 4 / 2
varCopy = var
print(var + (varCopy-2))
```

Listing 5: Flexibility in Python

Believe it or not, the first line runs just fine. In fact, Python handles order of operations, so it executes the computation, 4/2, before adding the 10. The resulting value of 'var' from the first line is thus 12. Then, on the second line, 'varCopy' is set equal to 'var'. So, 'varCopy' is equal to 12 as well. Finally, the print statement prints the resulting number from 'var+(varopy-2)' which reduces to '12+10'. The output of this code is $\boxed{22}$

$\Delta\Sigma$ Iterables and Sets

Consider, for a moment, the case when you need to create a computer program with lots of data. Obviously, you're probably not going to want to store 1000 distinct pieces of to 1000 distinct variables. This, is essentially why we have arrays, which are data structures that can contain massive amounts of data in one object. Such a structure can be indexed, and is therefore ordered. Data structures that have indices are known as **iterables**. The advantage to iterables are plentiful, but here are some of the most important motivations:

- Referencing data stored on an iterable is not syntactically complex.
- Running operations upon mass groups of data is easy.
- Iterables can contain other iterables, meaning they can represent $\mathbb{R}^n$ space for $n \geq 1$. This property is critical for all applied computer science including machine learning.

Here are some examples of different iterables in the Python programming language:

```
myList = [1, "cats", True, 3.14]
print(myList[0])   # we print the first element of 'myList'
myList[3] = myList[3] + 2   # here, we increment the fourth entry of 'myList' by two
print(myList[0] + myList[3])   # output is 6.14
print(myList[-1])   # prints the last element of 'myList' which is 3.14
```

Listing 6: The List

```
myTuple = (1, "cats", True, 3.14)   # here, we use paranthesis instead of brackets
print(myTuple[0])   # we print the first element of 'myTuple'
myTuple[3] = myTuple[3] + 2   # this line throws an error because tuple elements are
    unchangeable
```

Listing 7: The Tuple

```
myDict = {"username" : "codeKid_2003", "password" : "Tacocats!23"}   # Each entry has a key
    and a value. The ':' is like the '=' operator in this context. Entrys are seperated by
    commas
print(myDict["username"])   # output will say 'codeKid_2003'
```

Listing 8: The Dictionary

```
myString = 'abc'   # recall that a string is any sequence of characters encased by quotes.
    Strings are indexed as well
print(myString[2] + 'def')   # output will be 'cdef' which is the concatenation of 'c' and '
    def'
```

Listing 9: The String

You may also come across a case when you really don't need to store information in an ordered fashion. Instead, you may just need a **place** to store your data. In this case, you can use a **set**. A set is not indexed, meaning you can't reference specific objects in a set, rather, you can only know what is in and not in the set. This also means that you can access information such as the number of elements in the set. If you are familiar with mathematical set theory, then sets are identical in computer science. Therefore, there can be no repeats in a set. Here is an example of a set:

```
mySet = {1, 1, "hello", True, 2+2}
print(len(mySet))   # output is 4, because there are four distinct elements in the set
print(mySet[1])   # throws an error because sets are not indexed
```

Listing 10: The Set

ΔΣ Important Iterable and Set Methods

In a moment, you will get to see a collection of iterable methods, where some are specific to the iterable data type. However, one of the most important, universal lines of code you can run upon an iterable is the length method. We recently saw this in the context of a set, but here is how the len() method can be used for other iterables:

```
myList = [1, 2, 3, 4, 5]
myTuple = (1,2)
myDict = {"p1" : 1, "p2" : 2, "p3" : 3}
print(len(myList))   # output will return 5, because there are 5 elements in 'myList'
print(len(myTuple))   # output will return 2, because there are 2 elements in 'myTuple'
print(len(myDict))   # output will return 3, because there are 3 key-value pairs in 'myDict'
```

Listing 11: Length Method

```
myList.append(x)   # adds a new element, x, at the end of 'myList'
myList.count(x)   # returns the amount of elements, x, that occur in 'myList'
myList.insert(x, y)   # inserts a new element, y, at the specified index, x
myList.pop(x, y)   # removes an element, y, at the specified index, x
myList.remove(x)   # removes the first element, x, in 'myList'
myList.reverse()   # reverses the order of elements in 'myList'
myList.sort()   # sorts the elements in 'myList' from least to greatest
```

Listing 12: List Methods

```
myTuple.count(x)   # returns the amount of elements, x, that occur in 'myTuple'
myTuple.index(x)   # returns the index of the first appearance of the value, x, in 'myTuple'
```

Listing 13: Tuple Methods

```
myDict.keys()   # returns a list containing the keys of 'myDict'
myDict.values()   # returns a list containing the values of 'myDict'
myDict.setDefault(x)   # returns the value of the specified key, x. If the key, x, does not
    exist, then this method inserts the key, with the specified value, x
```

Listing 14: Dictionary Methods

```
myString.capitalize()   # returns 'myString' where myString[0] is capitalized
myString.isalnum()   # returns True if 'myString' contains strictly alphanumeric characters
    and False if otherwise
myString.isdigit()   # returns True if 'myString' contains strictly numerical characters and
     False if otherwise
"x".join(myIterable)   # returns a string of each element in 'myIterable' with 'x' in
    between
myString.split(x)   # returns a list of elements in myString that are seperated by a
    substring, 'x'
```

Listing 15: String Methods

```
mySet.add(x)   # stores the value, x, in 'mySet'
mySet.difference(x)   # (x represents a set) returns another set containing the elements
    dinstinct in 'mySet' and 'x'
mySet.union(x)   # (x represents a set) returns the union of 'mySet' and 'x'
mySet.intersection(x)   # (x represents a set) returns the intersection of 'mySet' and 'x'
mySet.issuperset(x)   # (x represents a set) returns True if 'x' is a superset of 'mySet'
    and False if otherwise
mySet.issubset(x)   # (x represents a set) returns True if 'x' is a subset of 'mySet' and
    False if otherwise
```

Listing 16: Set Methods

ΔΣ Slicing Iterables

Sometimes, you may want to reference only a certain grouping of an iterable, such as the first n elements where $n < \text{len}(x)$ and x is an iterable. In this case, slicing is very important. Slicing allows you to grab values from an iterable from a specified starting point to a specified ending point by a specified increment. Consider the following example:

```
myList = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
print(myList[0:3:1])   # returns 'myList' from the first element up **until** the fourth
    element by index increments of 1. So, the output in this case will be "[2, 4, 6, 8]"
print(myList[-8:-1:2])   # returns 'myList' from the 8th to last element up until the last
    element by index increments of 2. So, the output in this case will be "[6, 10, 14, 18]"
```

Listing 17: Slicing

ΔΣ Multidimensional Iterables

You may remember from the section, "Iterables and Sets" that iterables can be stored inside of other iterables. This implies that lists can be stored within lists, dictionaries inside of tuples, and strings inside of sets. We've actually already seen an instance of such a variable when we stored strings inside of lists. How exactly do we reference information inside of such variables? For the answer to this question, consider the following code:

```
myList = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]
```

Listing 18: Multidimensional List Example

Problem 2 (*Check Your Understanding*) What is the value of len(myList)?

Solution 2 (*Check Your Understanding*) The answer is 3, because there are 3 distinct lists inside of 'myList.'

This answer may or may not seem suprising weather or not you thought the answer was 9. While there are 3 lists of length 3 which multiples to 6, we only concern ourselves with the count of objects inside of 'myList,' which is the 3 lists.

Problem 3 (*Check Your Understanding*) What is the value of len(myList[0])?

Solution 3 (*Check Your Understanding*) Upon observation, we see that myList[0] = $[1, 2, 3]$ (do you see why?) Therefore, len(myList[0]) = 3 because there are 3 integers inside of myList[0]

This answer is **very important** to understand because it shows you how you can treat variable references as the objects they truly are. Specifically, we know that myList[0] = $[1, 2, 3]$, so running a code such as

```
print(myList[0][2])
```

Listing 19: Object Flexibility

is no different than asking the computer what the third element of $[1, 2, 3]$ is, which is just 3. To be truly fluent in Python means to be able to comfortably abstract information like this. You should always think about what certain variable references actually are as a data type or structure so you can properly deal with them. You have have lot's of practice opportunity in the 'Guess That Output' section of this handout.

ΔΣ Type Casting

Let's say you have two variables 'var1' and 'var2' defined as the following:

```
var1 = "3"
var2 = 4
```

Listing 20: Example Variable Deceleration

For some unspecified reason, you want to add these two variables together, so you run the following code:

```
print(var1 + var2)
```

Listing 21: Adding Numerical Values of Different Data Type

Unfortunately, this code throws you an error, telling you that you are prevented from adding integers to strings. Even though you should be able to just add 3 and 4, the datatypes are not compatible under the addition operator. To fix this, you can cast 'var1' to an integer so you are adding two integers:

```
print(int(var1) + var2)   # output will return 7, because the integer value of 3 plus the
    integer value of 4 is 7
```

Listing 22: Integers Casting Example

Type casting allows you to change the data type of an input. However, there are cases when a piece of data may not be transferable to a different datatype. Let's first look at the different casting methods in Python:

```
int(x)   # takes an input value, x, and casts it to an integer
str(x)   # takes an input value, x, and casts it to a string
float(x)   # takes an input value, x, and casts it to a float
bool(x)   # takes an input value, x, and casts it to a boolean
list(x)   # takes an input value, x, and casts it to a list
tuple(x)   # takes an input value, x, and casts it to a tuple
dict(x)   # takes an input value, x, and casts it to a dictionary
set(x)   # takes an input value, x, and casts it to a set
```

Listing 23: Casting Methods

Now, let's consider some cases where an input value for a casting method is incompatible with the proposed type cast:

```
int("abc")   # obviously, the characters inside the quotations are not strictly numerical
    digits, so this doesn't work
float([1,2,3])   # the input value for float() is simply not a number, so this doesn't work
```

Listing 24: Casting Methods

There are so many different combinations of cast-methods and data types to put together. The list is so exhaustive that it would make more sense to experiment than to go over each individual compatibility case.

ΔΣ Taking Inputs

So far, you likely only know how to initiate variables, manipulate values, and pass data to the console. Programming becomes genuinely interesting once you can create programs that autonomously handle tasks using logical instructions. Sometimes, the first step of this obtaining actual data from the user, which is why you can take inputs. Upon running the following code:

```
name = input("what is your name?: ")
print(name)
```

Listing 25: Input Example

your console will prompt you with the question, "what is your name?: " Then, you will have the ability to click and type a response in the console. When you press the enter key, whatever you typed in the console will be saved as a string to the variable, 'name'. Then, you will notice that your console repeats whatever you typed in the console. This is because the data you provided was stored in a variable, and then printed to the console. What's most important here is not the fact that you can even take inputs, it's instead the fact that the input is **always** stored as a string. So, let's say you want to take in two numbers as inputs to add together. You decide to write the following program:

```
num1 = input("First number: ")
num2 = input("Second number: ")
print(num1 + num2)
```

Listing 26: Input Addition Example

You will notice that after inputting '23' for the first number and '37' for the second number returns an output of '2337,' which isn't necessarily $23 + 37$. What went wrong? You forgot to cast the input values to integers so they can be added as integers instead of strings. The fixed program looks like this:

```
num1 = int(input("First number: "))
num2 = int(input("Second number: "))
print(num1 + num2)
```

Listing 27: Corrected Input Addition Example

Now, when you supply the input value for the 'num1,' the result is immediatley casted to an integer (asuming the input string is strictly numerical digits), and the same follows for 'num2.' Running your program with the same inputs yields the answer you wanted to begin with, which was $23 + 37 = 60$.

ΔΣ Printing the Type of a Variable

Sometimes, your code will become so complex that you yourself don't even know what your variable is equal to. This is usually not because you forgot, but instead, because your variable is being controlled autonomously somewhere in the code. If you need to find out the data type of your variable, you can run the following 'type()' method:

```
print(type(myVar))
```

Listing 28: Type Method

ΔΣ Guess That Output

```
print(str([0,1,2,[0,1,2,3],4][3][3]) + "6")
print(" ".join(['hello', 'my', 'name', 'is', 'corey']))
```

Listing 29: GTO #1

```
a = set(tuple([2,4,6,8]))
print(bool(len(a) * (1/len(a)) - 1))
```

Listing 30: GTO #2

```
print(str({1,2}.issubset({1,2,3,4})) + str(bool(0)))
print(str(bool(bool((bool(0))))) + str(bool(bool(bool(1)))))
```

Listing 31: GTO #3

```
a = [1, 3, 2, 4, 3, 5, 4, 6]
a.sort(); a.reverse()
print(a[0] - a[-1])
```

Listing 32: GTO #4

```
nums = [3.5, 4.5]
print(nums[0] + nums[1] - (int(nums[-1]) + int(nums[-2])))
```

Listing 33: GTO #5

ΔΣ CAPS

Problem 4 () Write a computer program that takes in an input and stores it to a variable called, 'name'. Then, using a print statement, make the computer greet the name that was initially taken as an input.

Problem 5 () Suppose $k = 2$ and $n = k^k$. What is the value of

$$\frac{n!}{k} + (n-k)^k$$

Problem 6 () Write a computer program that takes integer inputs for variables a and b, which denote the non-hypotenuse sides of a right triangle. Using a print statement, output the length of the hypotenuse. Repeat this three times and store each resulting hypotenuse in a list. Finally, sort the list and print the greatest value from among them.

(Note) you will need the first line to say 'import math' so you can run 'math.sqrt(x)' which returns the square root of the inout value, x.

Problem 7 () What is the value of

$$(1+2^{-1/32})(1+2^{-1/16})(1+2^{-1/8})(1+2^{-1/4})(1+2^{-1/2})$$

Problem 8 () Write a computer program that stores three lines of input values separated by spaces into three different variables. Store these variables as lists of integers in another list. This means you will have lists inside of a list. Then sort each list. Return the sum of the maximum values from each sub-list.

Problem 9 () Denote by p_k the kth prime number. Show that $p_1 p_2 \cdots p_n + 1$ cannot be the perfect square of an integer.

Problem 10 () Prove that it is impossible for three consecutive squares to sum to another perfect square.

Problem 11 () Store a line of inputs separated by spaces as integers into a list. Normalize the list.

(Bonus *) Find a non-zero list that is orthogonal to your list.

(Note) This problem requires the usage of generators which is not in the contents of this document.

ΔΣ Solutions to GTO

Solution 4 (*GTO 1*) We first find the third index of the list [0, 1, 2, [0,1,2,3], 4], which is the fourth element, which is [0,1,2,3]. Then, we find the third index of [0,1,2,3], which is 3. Then, we cast 3 to a string so "3" + "6" = "36". Thus, the first line prints $\boxed{36}$. The next line joins the sentence together into a string, where each element of the list is seperated by a space. Therefore, the second line prints $\boxed{\text{hello my name is corey}}$.

Solution 5 *(GTO 2)* In the first line, a list is casted to a tuple, which is then casted to a set. The set is given by $\{2, 4, 6, 8\}$. On the next line, we multiply the length of the set, which is 4, by $\frac{1}{4}$ (do you see why?) Since, $\frac{4}{4} = 1$, we subtract 1 to get 0. Casting this to a bool prints $\boxed{\text{False}}$ to the console.

Solution 6 *(GTO 3)* On the first line, we run $\{1, 2\}$.issubset($\{1, 2, 3, 4\}$), which is true since $\{1, 2\} \subset \{1, 2, 3, 4\}$. Then, we cast the 'True' keyword to a string and concatenate that the result of "str(bool((0))", which is 'False'. So the first line outputs $\boxed{\text{TrueFalse}}$ to the console. On the second line, the repetetive bool() methods don't continuously alter the input since you're casting the same data type to itself. So we can treat this line like it says 'str(bool(0))+str(bool(1))', which is just $\boxed{\text{FalseTrue}}$.

Solution 7 *(GTO 4)* On the first line, we define a list 'a'. On the second line, we sort the list and then reverse the sorted list. Since the 'a.sort()' method sorts 'a' from least to greatest, the reverse method leaves the list in order from greatest to least. Now, note that a[0] = 6 and a[-1] = 1 (make sure you see why). So, our output is $\boxed{5}$.

Solution 8 *(GTO 5)* On the first line, we define a list 'nums'. Note that nums[0] = 3.5, nums[1] = 4.5, nums[-1] = 4.5, and nums[-2] = 3.5 (again, make sure you understand!) Now, note that int(3.5) = 3 and int(4.5) = 4. So, our output is $\boxed{1}$.