**CS5001**
**Spring 2020**
# **Handout: Variables and Expressions**

All programming languages must be able to:

1. remember things,
2. repeat things,
3. communicate, and
4. make decisions.

Variables are all about #1: remembering things. We have some value (a number, a word, a sentence, a list, etc.) that we want to remember, so we name it and stick it somewhere in the computer's memory. Now we can access that value again, or modify it, or use it in a calculation, and all we need to remember is the name we gave it.

In Python, we always need to **initialize** a variable before we can meaningfully use it. When we initialize a variable, we assign it a value. We can modify the value later on, but it needs to have one right away or it's useless to us.

```
age                 # uninitialized; not ok. NameError!
age = 41            # initialized; everything is fine.
```

Whenever you see an equals sign like this, it means assignment. We're assigning the value 40 to the variable age. We refer to the equals sign as an **assignment operator**, and we also say "age gets 40."

When we initialize a variable, a few things are going on at once. We the programmer give the variable a label (also called a "name") and assign a value to it.  When we initialize the variable, the computer finds a spot in memory for the variable to live.

The programmer and computer are both responsible for the **data type** of each variable. The value we choose *implicitly* indicates the data type; by assigning `age` the value 41, we're indicating that `age` is a whole number, and that's how the computer treats it. The data type is important because different data types require different amounts of memory, can perform different actions, etc.

| **Variable Component** | **Who's Responsible** |
|---|---|
| Data Type | Programmer + Computer |
| Value | Programmer |
| Label | Programmer |
| Memory location | Computer |

A variable's value can change anytime it appears on the left of an equals sign (the assignment operator). If there is a variable on the right of an assignment operator, its value might be used in a calculation, but the variable itself definitely won't be modified.

```
age = 1
birth_year = 1978
curr_year = 2019
age = curr_year - birth_year
```

In this example, we're building on `age` with the variables `birth_year` and `curr_year`, and implicitly informing the computer of their data types: both are integers. Then we assign to age the value `curr_year - birth_year.` The value of age changes in the last line, but the other two variables are merely used in the calculation and ***cannot possibly change*** because they're on the right-side of the assignment operator.

Although there are a few others hanging around, we'll be using the following 4 data types in this class:

| **Data Type** | **Initialize** |
|---|---|
| integer | x = 2 |
| float | y = 2.87 |
| boolean | b = True |
| string | name = 'Laney' |

**Arithmetic Operations**
Arithmetic Operations apply to numeric data types (float and int). As we just saw, arithmetic operations happen on the right-side of the assignment operator. The right-hand side is evaluated first, and then is assigned to whatever variable is on the left.

Python has your basic built-in arithmetic operators, plus a couple of fun ones you might not recognize.

| **Operator** | **Operation** | **Example** | **Value of x** |
|---|---|---|---|
| + | Addition | x = 20 + 15 | 35 |
| - | Subtraction | x = 20 - 15 | 5 |
| * | Multiplication | x = 20 * 30 | 600 |
| ** | Exponentiation | x = 2 ** 4 | 16 |
| / | Division | x = 20 / 15 | 1.3333 |

| | | | |
|---|---|---|---|
| `//` | Integer Division | `x = 20 // 15` | 1 |
| `%` | Modulo | `x = 20 / 15` | 5 |
| `+=` | Increase | `x = 1`<br>`x += 5` | 1<br>6 |
| `-=` | Decrease | `x = 1`<br>`x -= 1` | 1<br>0 |

**Integer Division (//) and Modulo (%)**

The “regular” division operator (/) works exactly like you’d expect. 20/15 turns out to be 1.33333.

The division operator (//) performs **integer division.** It’s a throwback to the long division we all learned in third grade, but I also like to think of it as *Price is Right* rules: you want to get as close as you can without going over.

When dividing 20 // 15, we know that 15 goes into 20 just once. 15*1 is 15, but 15*2 is 30, so we’d go over. Can’t do that on *The Price is Right*! That’s why **`x = 20 // 15`** assigns the value 1 to `x`.

The modulo operator is on the flip side. It’s the remainder leftover from doing integer division. When dividing 20 / 15, we know that 15 goes into 20 once, with a remainder of 5. That’s why **`x = 20 % 15`** assigns the value 5 to `x`.

**Increase (+=) and Decrease (-=) Operators**

You’ll see these shortcut-operators used often in Python and other programming languages. They’re just like addition and subtraction, but because they’re shorter and more concise, we programmers like them -- less work for us.

These two code snippets are equivalent:

| | |
|---|---|
| `x = 0`<br>`x = x + 1` | `x = 0`<br>`x += 1` |

So are these:

| | |
|---|---|
| `x = 0`<br>`x = x - 1` | `x = 0`<br>`x -= 1` |