Week 1:
Value Types, Operations, Print Statements, and Variables

Value Types

Integers

Integers are whole numbers, or numbers without a decimal part. When we type negative numbers, we will put them in parentheses. (The computer won't do this with its output! But it will make our code easier to read.)

Examples are 5, (-10), 0, and 999999999.

Floats

Floats are numbers that have a decimal part (at least one number after the decimal point). This decimal part can be zero, but it must be included. To a computer, 5 and 5.0 are different things - the first one is an **integer** and the second one is a **float**. Just like with integers, negative floats should be put in parentheses.

Examples are 5.2, (-1.4), 0.0, and 0.0000001

Characters

A **character** is a single printed (but not always visible!) letter or symbol. Characters can be numbers, so to keep from confusing **characters** with **integers**, we put characters in single quotes. To a computer, the **character** '5' is different from the **integer** 5. How are they different? Computers can only add integers and floats, and can only display characters, so if you ask a computer what '5' + '7' is, it will throw an error. Similarly, if you ask it to display the integer 5, it will also throw an error[1].

The character ‘a’ is also different from the character ‘A’ to a computer. Internally, characters are stored as 1’s and 0’s, so even though ‘a’ and ‘A’ look similar to us, a computer might store them in **ASCII** format, where they would look like this:

```
'a' -> 01100001
'A' -> 01000001
# Those are totally different numbers!
```

Examples of characters are ‘b’, ‘B’, ‘5’, ‘%’, and ‘ ’ (The space character! We will talk more about invisible, or “whitespace”, characters later on.)

Strings

A **string** is a list of zero or more **characters**, commonly words, sentences, or other text. To make sure we don’t confuse **strings** and **characters**, we will put strings in double quotes.

Since strings are made up of characters, they can be displayed by computers. (This is good, because it’s much easier to ask the computer to display the string “Hey!” than to ask it to display the character ‘H’, then the character ‘e’, then ‘y’, then ‘!’)

The fact they are made up of characters also means strings are case-sensitive, just like characters. The string “hello” and the string “Hello” are totally different things to a computer, even though a human can tell they mean the same thing.

Strings can sometimes be only one character long. To a computer, the **string** “5” and the **character** ‘5’ are different things. How are they different? Well, the **string** “5” *contains* the **character** ‘5’, just like the **string** “ab” *contains* the **characters** ‘a’ and ‘b’.

Strings can also be zero **characters** long, which is known as **the empty string**, and looks like this: “”

Just like **characters**, strings can look like **integers** and **floats**. To help remember the difference, imagine that **strings** are being written on a piece of paper, while **integers** and **floats** are being typed into a calculator. If you write “2+5” on your paper, the numbers will not be added together. By contrast, if you put 2+5 into your calculator, then you can add them!

Examples of **strings** include “howard”, “Howard”, “a”, “This is a string, too!”, “5”, “-2.4”, “0.0” and “” (**The empty string**)

Booleans

Booleans are the simplest value. They contain only two values: **True** and **False**. Why save them for last if they are simplest? Well, remember computers case about capitalization - if you write true instead of **True**, or false instead of **False**, the computer will not understand you!

Values

Collectively, we refer to **booleans**, **integers**, **floats**, **characters**, and **strings** as **value types**.

Operations

Integer Operations

Addition

Just like you remember from highschool, you can add integers. We use the ‘+’ character for addition.

Example Problems: (The ‘=’ character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use “->” instead to denote the result of operations.)

5 + 5 -> ____

(-8) + (-9) -> ____

0 + 0 -> ____

Subtraction

Subtraction also works just as you’d expect, and we use the ‘-’ character for it. If you need to subtract a negative number, you should put parentheses around the negative number.

Example Problems: (The ‘=’ character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use “->” instead to denote the result of operations.)

7 - 3 -> ____

3 - 7 -> ____

7 - 7 -> ____

(-3) - 7 -> ____

7 - (-3) -> ____

(-3) - (-7) -> ____

Multiplication

Multiplication is also straightforward. We use the '*' character to tell the computer to multiply. Using 'x' will not work!

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

7 * 7 -> ____

7 * 0 -> ____

(-7) * 7 -> ____

(-7) * (-7) -> ____

Exponential Powers

Computers can also calculate exponential powers on integers. If you need a refresher, an exponent simply tells you to multiply a number by itself a certain number of times. So, for instance: 3^2 says to multiply 3 by itself 2 times, and is equal to 3*3; 9^5 says to multiply 9 by itself 5 times, and is equal to 9*9*9*9*9. Python uses "**" to denote exponential powers.

(Special case: Any number raised to the 0'th power [as in 5^0] is 1.)

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

2**3	->	Equivalent multiplication: ____	->	Answer: ____
3**2	->	Equivalent multiplication: ____	->	Answer: ____
1**5	->	Equivalent multiplication: ____	->	Answer: ____
10**1	->	Equivalent multiplication: ____	->	Answer: ____
100**0	->	Equivalent multiplication: ____	->	Answer: ____
0**10	->	Equivalent multiplication: ____	->	Answer: ____
0**0	->	Equivalent multiplication: ____	->	Answer: ____

Division

Division is **not** straightforward. When you divide 5 by 2, you would normally expect to get 2.5. This is **not** how division works **with integers**. (Floats are a different matter!) Division on integers uses a special type of division called **integer division**. Python uses "//" to denote integer division. Do not use '/' for integers![2]

Integer division will take the result of the division, and **round down** to the nearest integer, like below:

```
>>> 12 // 5
2
```

Integers won't change, since they are already integers:

```
>>> 15 // 5
3
```

For negative numbers, remember that rounding **down** means that you need the **value** of the number to be smaller, not for the **digits** to be smaller. So a division that resulted in -7.5 would round **down** to -8, rather than **up** to -7, since -8 is the closest integer that is **smaller** than -7.5:

```
>>> (-15) // 2
-8
```

Just like in normal division, you can't divide by 0! If you try to, the computer will throw an error that looks like this:

```
ZeroDivisionError: integer division or modulo by zero
```

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

3 // 1	->	____	
1 // 3	->	____	
(-5) // 2	->	____	# Note: Remember to round **down**!
(-5) // (-2)	->	____	
12 // 0	->	____	
0 // 12	->	____	
0 // 0	->	____	
7 // 7	->	____	

Modulo

Modulo is an operation that might be entirely new to you! Modulo tells you the **remainder** of a division operation. We represent modulo with '%'.

To calculate the modulo, find the first number **smaller** than or equal to the **dividend** (the number on the left of the '%') that is evenly divisible by the **divisor** (the number on the right of the '%'). Once you have that number, the modulo is simply the difference between that number and the dividend!

- For 11 % 5
    - The **dividend** is ____
    - The **divisor** is ____
    - The first number smaller than or equal to the dividend that is evenly divisible by the divisor is ____
    - The result of the modulo is ____

- For 10 % 5
    - The **dividend** is ____
    - The **divisor** is ____
    - The first number smaller than or equal to the dividend that is evenly divisible by the divisor is ____
    - The result of the modulo is ____
- For 9 % 5
    - The **dividend** is ____
    - The **divisor** is ____
    - The first number smaller than or equal to the dividend that is evenly divisible by the divisor is ____
    - The result of the modulo is ____
- For 4 % 5
    - The **dividend** is ____
    - The **divisor** is ____
    - The first number smaller than or equal to the dividend that is evenly divisible by the divisor is ____
    - The result of the modulo is ____
- For (-4) % 5
    - The **dividend** is ____
    - The **divisor** is ____
    - The first number **smaller** (remember, this means its **value** is less!) than or equal to the dividend that is evenly divisible by the divisor is ____
    - The result of the modulo is ____

- For 4 % (-5)
    - The **dividend** is ____
    - The **divisor** is ____
    - The first number smaller than or equal to the dividend that is evenly divisible by the divisor is ____
    - The result of the modulo is ____
- For (-4) % (-5)
    - The **dividend** is ____
    - The **divisor** is ____
    - The first number **smaller** than or equal to the dividend that is evenly divisible by the divisor is ____
    - The result of the modulo is ____

You can't modulo by 0 either, just like you can't divide by zero. If you try to, the computer will throw the same error as with integer division. So if you typed 7 % 0, you would see this:

```
ZeroDivisionError: integer division or modulo by zero
```

More Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

17 % 5 -> ____

17 % 17 -> ____

17 % 0 -> ____

0 % 17 -> ____

(-11) % 5 -> ____

(-11) % (-5) -> ____

11 % (-5) -> ____

0 % 0 -> ____

-1 % -1 -> ____

Float Operations

Addition

Addition for floats works just like for integers, and just like you're used to from highschool. We use '+' for float addition, just like for integers.

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

5.5 + 6.1 -> ____

(-0.1) + 0.0 -> ____

9.99 + (-0.4) -> ____

Subtraction

Similarly to addition, subtraction on floats works just like in highschool. We use '-' for float subtraction, just like for integers.

Example Problems:

5.5 - 6.1 -> ____

(-0.1) - 0.0 -> ____

9.99 - (-0.4) -> ____

Multiplication

Multiplication is also straightforward. We use '*' for float multiplication, just like for integers.

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

1.0 * 4.3 -> ____

10.0 * (-4.3) -> ____

(-10.0) * (-4.3) -> ____

Exponential Powers

Exponential powers follow the same pattern as with integers, as long as the power (the 3 in 5^3, or the number to the right of the "**") is an integer. So 1.5**3 is equal to 1.5*1.5*1.5. When the power is a float, exponents get more complicated. You're welcome to look into this, or talk to me about it after class, but you will not be required to use exponential powers where the power is a float.

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

1.5**3 -> Equivalent multiplication: ____ -> Answer: ____

1.5**0 -> Equivalent multiplication: ____ -> Answer: ____

0.0**0 -> Equivalent multiplication: ____ -> Answer: ____

Division

Believe it or not, floating point division is actually easier than integer division! Float division is the way computers do division that is like we are used to from highschool. We use '/' for float division. Just like with integer division, you cannot divide by zero.

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

10.0 / 2.5 -> ____

10.0 / (-2.5) -> ____

(-10.0) / 2.5 -> ____

(-10.0) / (-2.5) -> ____

2.5/0.0 -> ____

0.0 / 0.0 -> ____

Modulo

Modulo on floating point numbers in Python works just like modulo on integers. However, not all languages support modulus on floating point numbers, and we will not be using it in this course. Feel free to mess around with it and try it out! Just know that you will not need it for a graded assignment.

String Operations

Concatenation

We can **concatenate** two strings with the '+' symbol in Python. Concatenation takes the second string, and appends it to the first string, so "Hey" + "!" will become "Hey!".

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

"abc" + "def" -> _____

"hello " + "there" -> _____

"howard" + "" -> _____

"" + "" -> _____

String Multiplication (Advanced)

String multiplication is a special Python-only operation that uses the '*' character. To use string multiplication, you multiply a string and an integer N together. The result is a new string that

consists of your original string repeated N times. For example, to repeat the string “a” five times, we would write `“a” * 5`.

Note that if you multiply a string by a number that is zero or negative, you get back the empty string!

Example Problems: (The ‘=’ character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use “->” instead to denote the result of operations.)

“ab” * 3 -> _____

“A” * 4 -> _____

“A ” * 4 -> _____ # Note the space after the ‘A’ character!

“HELLO” * 2 -> _____

“HELLO” * 1 -> _____

“HELLO” * 0 -> _____

“HELLO” * -1 -> _____

“” * 5 -> _____

Boolean Operations

and

The `and` operation will output True if and only if **both** inputs are True. In Python, we use the word `and` for the `and` operation. (Remember that computers are case sensitive, so `AND` won’t work!)

Example Problems: (The ‘=’ character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use “->” instead to denote the result of operations.)

True `and` False -> _____

True `and` True -> _____

False `and` True -> _____

False `and` False -> _____

or

The `or` operation will output True if **either** input is True. Compare this with `and`, where **both** inputs have to be True. In Python, we use the word `or` for the `or` operation. (Remember that computers are case sensitive, so `OR` won't work!)

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

True `or` False -> _____

True `or` True -> _____

False `or` True -> _____

False `or` False -> _____

not

The `not` operation reverses the boolean value that comes after it, so True becomes False, and vice-versa. In Python, we use the word `not` for the `not` operation. (Remember that computers are case sensitive, so `not` won't work!)

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

`not` True -> _____

`not` False -> _____

Multiple Operations at Once

Sometimes we want to be able to ask a computer to do more than one thing at once. Maybe we want to know what 5 + 4 + 3 is, or to **concatenate** "ab" + "cd" + "ef".

Multiple Operations on Numbers

When working on numbers (**integers** and **floats**), operations are carried out in **PEMDAS** order. If you need a brief refresher on PEMDAS, it stands for "Parentheses, Exponents, Multiplication/Division, and Addition/Subtraction". Here's how it works:

1) Check for parentheses. If there are any, find all the top-level parenthetical groups (a top-level parenthetical group is one that is not contained inside any other parenthetical groups). From left to right, take each group, including any nested parenthetical groups, perform PEMDAS on it, then replace the entire parenthetical group with the answer.
2) Once there are no parentheses, check if there are any exponents. If there are, perform them from left to right.
3) Once there are no exponents, check if there are any multiplications or divisions. If there are, perform them from left to right.
4) Once there are no more multiplications or divisions, check if there are any additions or subtractions. If there are, perform them from left to right.
5) Once you've done this, you'll have only one number, and you're done!

This means that if you have an expression like this:

```
(5 + (2 + 3)) // 1 ** 4 - 5 * 3
```

It is evaluated in the following way:

1) **P**EMDAS - check for parentheses. We have some, so we take the expression in the **outermost** set of parentheses, `5 + (2 + 3)`, and evaluate it.
    a) **P**EMDAS - check `5 + (2 + 3)` for parentheses. We still have some, so we take the expression in the outermost (and only!) set of parentheses, `2 + 3`, and evaluate it.
        i) **P**EMDAS - check `2 + 3` for parentheses. There are none.
        ii) P**E**MDAS - check `2 + 3` for exponents. There are none.
        iii) PE**MD**AS - check `2 + 3` for multiplications and divisions. There are none.
        iv) PEMD**AS** - check `2 + 3` for additions and subtractions. There is one addition, so we perform it and get `5`.
        v) Now that we have a single number, we're done!
    b) We replace everything inside the parentheses we just evaluated with the result, so `(2 + 3)` becomes `5`, and our expression is now `5 + 5`.
    c) **P**EMDAS - check `5 + 5` for parentheses. (We already checked once, but we can't move on to the next letter until we're sure there's nothing else to do for this one!) There are none.
    d) P**E**MDAS - check `5 + 5` for exponents. There are none.
    e) PE**MD**AS - check `5 + 5` for multiplications and divisions. There are none.
    f) PEMD**AS** - check `5 + 5` for additions and subtractions. There is one addition, so we perform it and get `10`.

    g) Now that we have a single number, we're done!
2) We replace everything inside the parentheses we just evaluated with the result, so `(5 + (2 + 3))` becomes `10`, and our expression is now `10 // 1 ** 4 - 5 * 3`.
3) **P**EMDAS - check `10 // 1 ** 4 - 5 * 3` for parentheses (We already checked once, but we can't move on to the next letter until we're sure there's nothing else to do for this one!) There are none.
4) P**E**MDAS - check `10 // 2 ** 4 - 5 * 3` for exponents. There is one, so we calculate `2**4`, which is `16`, so our expression becomes `10 // 16 - 5 * 3`.
5) P**E**MDAS - check `10 // 16 - 5 * 3` for exponents. There are none.
6) PE**MD**AS - check `10 // 16 - 5 * 3` for multiplications and divisions. There are two, so we do the one on the left, and calculate `10 // 16`, which is `0`, so our expression becomes `0 - 5 * 3`.
7) PE**MD**AS - check `0 - 5 * 3` for multiplications and divisions. There is one multiplication, so we calculate `5 * 3`, which is `15`, so our expression becomes `0 - 15`.
8) PE**MD**AS - check `0 - 15` for multiplications and divisions. There are none.
9) PEMD**AS** - check `0 - 15` for additions and subtractions. There is one subtraction, so we calculate `0 - 15`, and get `-15`.
10) PEMD**AS** - check `-15` for additions and subtractions. There are none.
11) Now we have reached the end of PEMDAS, and we have a single number left, so we are done, and our answer is `-15`!

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

5 + 6 * 3	->	____
5.0 + 6.0 * 3.0	->	____
(5.0 + 6.0) * 3.0	->	____
2 // 2 * 0	->	____
2 // (2 * 0)	->	____
4 * 3 ** 2	->	____
3 ** 2 * 4	->	____
2 ** 2 * 2	->	____
2 ** (2 * 2)	->	____

(5 + 6 * 2) // 5 ** 2 -> ____

2 * (2 * (2 * (1 + 1))) -> ____

(2 ** (2 * 2) // 2) * 2 -> ____

Scratch Paper:

Multiple Operations on Strings

Concatenation

Multiple strings can be concatenated at once! They are simply all merged together into one long string, so `“ab” + “cd” + “ef”` becomes `“abcdef”`.

Example Problems: (The ‘=’ character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use “->” instead to denote the result of operations.)

“ho” + “wa” “rd”	->	_____
“no” + “spaces” + “here”	->	_____
“space” + “ “ + “character”	->	_____
“” + “” + “”	->	_____

String Multiplication (Advanced)

String multiplication can be combined with other operations in two ways.

1) It can be included in concatenations, to concatenate multiple copies of a string. An example of this would be `“a” + “b” * 3 + “c”`, which would result in `“abbbc”`.
2) Mathematical expressions can be used in the place of a single integer, so the expression `“howard” * (4 // 2)` would result in `“howardhoward”`. Note that, for PEMDAS purposes, string multiplication has the same precedence as normal multiplication. To make your code safer and more readable though, you should always put parentheses around a mathematical expression if you’re using it for string multiplication! So you should write `“howard” * (2 ** 2)` instead of `“howard * 2 ** 2`, even though the latter is technically valid.

Example Problems: (The ‘=’ character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use “->” instead to denote the result of operations.)

“a” * 3 + “b” * 2	->	_____
“ho” + “wa” * 3 + “rd”	->	_____
“ho” + “wa” * (12 // 4) + “rd”	->	_____
“a” * (4 + (3 // 2 ** 2) * 5 // 1) + “b” * (3 // 2)	->	_____

Boolean Operations

Boolean operations work according to a system similar to PEMDAS. For booleans, the ordering is PNAO, which is unfortunately not nearly as catchy. You can try to remember it by remembering the sentence “Pets need amazing owners”.

PNAO stands for parentheses, not, and, or. Just like with PEMDAS, we perform boolean expressions in a particular order:

1) Anything in parentheses first, going from left to right if there are multiple top-level parenthetical groups.
2) We apply all `not`s, going from left to right if there are multiple.
3) Then we do our `and`s, again working from left to right.
4) Finally we do `or`s left to right as well.

Example Problems: (The ‘=’ character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use “->” instead to denote the result of operations.)

True `and` False `or` True	->	_____
True `and` (False `or` True)	->	_____
True `and not` False	->	_____
True `or` False `and` True	->	_____
True `and not` False `or` False	->	_____
`not not` False	->	_____
True `and` True `and` False	->	_____
False `or` True `or` False	->	_____
False `and` True `and` True `or` False `or` True	->	_____
False `and` True `and` (True `or` False `or` True)	->	_____
`not not` False `and` (True `or not` (False `or` True))	->	_____

The Print Statement

The Print Function

The `print` function in Python consists of the word `print`, followed by an opening parenthesis '(', a **string**[3], and a close parenthesis ')'.

Print will take the **string** in between the parentheses and display it on the screen.

Example Problems:	Output:
`print("Hello, world!")`	"Hello, world!"
`print("some words")`	_____
`print("12345")`	_____
`print("5")`	_____
`print("")`	_____

Value Conversion

If the print statement can only print strings[3], how do we see the result of operations on numbers? To handle this, Python provides **functions** that will convert data from one type to another. **Functions** are simply a special word followed by parentheses, like the `print` function, that does something; we will cover functions in more detail later, so don't feel like you need to understand them now. All you need to know for now is how to use the following conversion functions:

`int`: To convert something to an integer, simply use the `int` function. This is done by writing `int`, then an opening parenthesis '(', then the value to convert, then a closing parenthesis ')'. For example, to convert the string "12" into the integer 12, we would write `int("12")`, which would give `12`. You can also convert floats, but the `int` function will **truncate**, or chop off, everything past the decimal point. So `int(1.9)` will give `1`.

`float`: This function works just like `int`. Simply type `float`, then an opening parenthesis '(', then a value to convert, then a closing parenthesis ')'. So `float("12.3")` would give us `12.3`. You can also convert integers to floats, and the function will simply add a `.0` at the end to show it is now a float. So, for instance, `float(1)` would give `1.0`.

`str`: The `str` function converts things to strings. As with the other functions, to use the `str` function we simply write `str`, an opening parenthesis '(', the value to convert, and then a closing parenthesis ')'. So `str(12)` gives `"12"`, and `str(12.1)` gives `"12.1"`.

Example Problems: (The '=' character has a special meaning in computer science, and does not mean the same thing as in highschool, so we will use "->" instead to denote the result of operations.)

str(10)	->	____
int("10")	->	____
float(10)	->	____
float("10")	->	____
int(10.0)	->	____
str(10.0)	->	____

Variables

Motivation

Suppose Alice and Bob go to a restaurant. When it comes time to pay, they decide that Alice will pay for the food, and Bob will each pay her back ½ of the total price. However, Alice only has the values of the individual dishes, and doesn't know the total. There were three dishes bought, one that cost $19.55, one that cost $21.05, and one that cost $5.50. In order to find out the total to pay, as well as the amount Bob should pay her back, Alice would have to do something like this:

```
print(str(19.55 + 21.05 + 5.50))  # The total value Alice has to pay
print(str( (19.55 + 21.05 + 5.50) / 2.0)) # The total value Bob will pay
```

This is inefficient though! Notice how we're adding `19.55 + 21.05 + 5.50` twice. **Variables** allow us to save values in order to use them later.

(The `#` character denotes a **comment**. Everything after a `#` character on a line is ignored by Python. This lets us leave notes for ourselves and for other programmers, just like how I explained the purpose of each line in the example above.)

How to Create a Variable

It's very easy to make a variable! All you have to do is write the **name** of the variable you want to create, followed by an **equals sign** (Remember how we couldn't use them in our example problems because they were special? This is why!), followed by the value you want to save.

Variable **names** can take many forms, but for our purposes we will always write them in the following way:

1) First, we write the **type** of the data we want to store in the variable. For now, this will be "integer", "float", "character", "string", or "boolean".
2) Next, we write an underscore: '_'
3) Finally, we write a descriptive name telling us what the variable will be used for, such as "counter", "name", or the like.

Note that although Python does not require variables to be named in this way, we will stick to this convention for a couple of reasons. First, when you move on to using another language in CS II, it will make things a lot easier for you. Second, this give you a better idea of how the computer actually works under the hood. Lastly, it just makes your code easier to read! Descriptive variable names mean you can look at the code and tell what it's doing with much less effort, and as programmers, we strive to be lazy at all times!

Now, although we will always use descriptive variable names in our code, you should be able to read code where the names are not descriptive. You can call a variable anything you want, and it will still work, but it will be harder to understand the code. For the exercises in this lesson, you will practice with some **un**descriptive variable names, to make sure you understand that the name truly can be anything. I expect to see descriptive, meaningful variable names in your code, however!

Some examples of undescriptive, but otherwise acceptable, variable names:

```
string_foo = "howard"
integer_bar = 12
float_baz = 54.321
boolean_bat = True
```

The variable `string_foo` now contains the string `"howard"`

The variable `integer_bar` contains the integer ____

The variable `float_baz` contains the float ____

The variable `boolean_bat` contains the boolean ____

A few last notes about variables:

1) The variable name **must** be on the left of the equals sign. If you put the **value** on the left and the **name** on the right, it will not work!
2) Remember, uppercase and lowercase letters are different to the computer! So the variables `string_foo` and `String_Foo` are different!

Example Problems:

string_somewords = “DC”	Variable Name: _____ Type Contained: _____ Value Contained: _____
string_wordsandnumbers123 = “Howard”	Variable Name: _____ Type Contained: _____ Value Contained: _____
integer_with_underscores_now = 151	Variable Name: _____ Type Contained: _____ Value Contained: _____
BOOLEAN_ALL_CAPITALS = True	Variable Name: _____ Type Contained: _____ Value Contained: _____
cHaRaCtEr_sOmEcApiTALs = ‘g’	Variable Name: _____

# Note: Although this is technically valid, you should not mix upper and lower case like this unless you have a good reason! It makes your code hard to read.	Type Contained: _____ Value Contained: _____
string_test = “remember that capitals” string_Test = “make a difference!”	1st Variable Name: _____ 1st Type Contained: _____ 1st Value Contained: _____ 2nd Variable Name: _____ 2nd Type Contained: _____ 2nd Value Contained: _____
float_i_hold_the_answer = 5.0 / 2.0	Variable Name: _____ Type Contained: _____ Value Contained: _____
integer_expression_result = (5 + 3) * 5 // 2	Variable Name: _____ Type Contained: _____ Value Contained: _____

How to Use a Variable

Using a variable is easy too! You can use a variable anywhere you would normally use a value, and the value contained in the variable will be used instead. So for instance:

```
string_foo = "Howard"
print(string_foo)
```

First, we save the value ”Howard” into the variable `string_foo`, then we call the function `print` with `string_foo` in-between the parentheses. Normally `print` need a string there, but

since we passed a variable with a string **value**, `print` is smart enough to simply use the value instead! So it will print out `"Howard"`.

Here's another example:

```
integer_foo = 12 + 34
string_foo = str(integer_foo)
print(string_foo)
```

In this case, we first save the result of adding `12 + 34` into `integer_foo`. Next, we call `str(integer_foo)`. The `str` function then looks at the value stored in `integer_foo`, and converts that **value** to a string. (Note that it **does not** convert the words "integer_foo" to a string! It converts the **value stored in** `integer_foo`.) Lastly, we simply print out `"46"`.

One more example:

```
integer_foo = 12
integer_bar = integer_foo + 34
string_bar = str(integer_bar)
print(string_bar)
```

This code outputs `"46"` just like the last example, but notice what we do on the second line. Instead of doing our calculation with just numbers, we have the `12` saved into the variable `integer_foo`, and then add `integer_foo + 34`. Just like how `str` and `print` look up the value when they are given a variable, so do all other operations! So since the value stored in `integer_foo` is `12`, the computer will compute `12 + 34`, and then store that value in `integer_bar`.

Hint: If you're having trouble remembering what the values of the variables are, you can write the values in the left margin next to the places where the variables are used, to remind yourself that we are using the **value**.

Example Problems: (Again, these do not all use descriptive names, but this is only because you should practice understanding what code is doing without descriptive variables. Your code's variable names should describe what the variable is doing.)

`string_tree = "some string"` `print(string_tree)`	What is printed?
`integer_number = 321` `string_number = str(integer_number)`	What is printed?

`print(string_number)`	
`integer_val = 43 + 21` `string_output = str(integer_val)` `print(string_output)`	What is printed?
`integer_number = 7 * 7` `string_number = str(integer_number)` `print(string_number)`	What is printed?
`string_foo = "concatenating"` `string_bar = string_foo + "strings"` `print(string_bar)`	What is printed?
`integer_foo = 7` `integer_bar = integer_foo * 3` `string_value = str(integer_bar)` `print(string_value)`	What is printed?
`boolean_obst = True` `string_kartoffel = str(boolean_obst)` `print(string_kartoffel)`	What is printed?
`integer_a = 3` `integer_b = 7` `integer_total = integer_a * integer_b` `string_to_print = str(integer_total)` `print(string_to_print)`	What is printed?

`` integer_rabbit = 35 * 2 integer_dog = integer_rabbit // 2 string_cow = str(integer_dog) print(string_cow) ```	What is printed? (Remember, the variable **names** don't matter! Only the **values** stored in them matter! Don't let the unusual names throw you off!)
`` string_ab = "cd" string_cd = "ab" string_ef = string_ab + string_cd print(string_ef) ```	What is printed? (Remember, the computer will use the **values** in the variables, not the **names** of the variables!)

Variable Reassignment

What do you think will be printed by the following code?

```
string_gato = "String One"
string_gato = "String Two"
print(string_gato)
```

Will it print "String One"? Or "String Two"? It turns out it will print "String Two". This is because whenever we assign a value to a variable (by writing the variable name, an equals sign, and then the value, such as `gato = "String Two"`) we **overwrite** whatever is in there previously. For example, imagine variables as a whiteboard. When we assign a value to them, we **erase** whatever was there previously, and put something new on the whiteboard instead. The old value is entirely gone! The variable will now have whatever the new value we assigned to it is.

Example Problems:

`` string_foo = "A" string_foo = "B" print(string_foo) ```	What is printed?
	What is printed?

`` string_foo = "A" string_bar = "B" print(string_foo) ```	
`` string_foo = "A" print(string_foo) string_foo = "B" ```	What is printed?
`` print(string_foo) string_foo = "A" string_foo = "B" ```	What is printed? (This one is a trick question!)

---

[1] This is not entirely true in Python, the language we will be using, for two reasons. First off, Python "cheats" a little bit, so characters, in Python, are actually strings of length one. However, this is not the case in every language, so we will always differentiate between characters and strings. Secondly, in Python you can add strings, but it happens via a process called **concatenation**, rather than the addition you are used to. This is covered more in the operations section of this handout, but the short version is that the two strings will be combined, so "5" + "7" becomes "57".

[2] While using '/' will work on integer values, you should get in the habit of always being clear about if you are using integer or floating point division (covered in the float operations section), so in this class we will always use "//" when working with integer values, so that we know we are doing integer division.

[3] This is not entirely true either. In Python, the language we will be using, `print` will accept **float** and **integer** values as well. However, this is an example of Python "cheating" - behind the scenes, Python is turning those **float** and **integer** values into **strings** for you. For this class, you should only pass **string** values to `print`.