Introduction to Computer Programming

Data Types, Errors and Debugging, Advance Math Operations & Formatting Output

CSCI-UA.0002-0010
Data Types

Python needs to know how to set aside memory in your computer based on what kind of information you want to store

There are three basic types of data that we will be working with during the first half of the term:

- Strings (Character-based Data)
- Numbers
- Logical Values (True / False)
Data types dictate types of data being stored.

Data types dictate the valid operations that can be performed on those values.
Numeric Data Types

Integers
- Whole numbers that do not contain a decimal point
- Abbreviated as “int” in Python
- Example: 5, -5, 100, 10032

Floating Point Numbers
- Numbers that contain a decimal point
- Abbreviated as “float” in Python
- Example: 5.0, -5.0, 100.99, 0.232132234
Numeric Data Types

You can store numeric data inside variables that you create.

Example:

```python
num_1 = 5  # this is an int
num_2 = 4.99  # this is a float
```

Keep in mind that you do not use separators or symbols when storing numeric data. We will use formatting to do this.

Example:

```python
num_3 = $5,123.99 # error!
```
What’s the data type?

5
5.5
"Hello"
"5.5"
2.975
2.0
Numeric Data Types

Python is not a strictly typed language. This means that you don’t need to pre-declare what kind of data your variables will be holding.

This is also called “dynamic typing”.
## Data Types Across Languages

<table>
<thead>
<tr>
<th>Loosely Typed</th>
<th>Strictly Typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>C</td>
</tr>
<tr>
<td>PHP</td>
<td>C++</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Java</td>
</tr>
<tr>
<td>Perl</td>
<td>ActionScript</td>
</tr>
</tbody>
</table>
Strictly Typed Languages - Examples

ActionScript

```
var name:String = "Harry";
var top_speed:Number = 50;
var gravity:Number = 9.5;
```

Java

```
String name = "Harry";
int top_speed = 50;
float gravity = 9.5;
```
User Input and Math Expressions

We can capture input from the user (via the input() function) and use that input in our calculations.

However, the input() function “returns” a string – this means that the data type that “comes out” of the input() function is a series of printed characters.

We need to convert the result of the input function from a string into one of the two numeric data types that Python supports (float and int).
**float(), int(), & str() Functions**

float(), int(), and str() functions are data type conversion functions. Each takes an argument and converts that argument into specified data types.
Example:

#ask the user for their monthly salary
monthly_salary = input('How much do you make in a month?')

#convert the salary into a float
monthly_salary_float = float(monthly_salary)

#calculate the yearly salary
yearly_salary = monthly_salary_float * 12

#display the results
print('That means you make', yearly_salary, 'in a year')
Example:

#ask the user for their monthly salary
monthly_salary = float(input('How much do you make in a month?'))

#calculate the yearly salary
yearly_salary = monthly_salary * 12

#display the results
print('That means you make', yearly_salary, 'in a year')
Nesting Data Type Conversions

```
monthly_salary = float(input('How much do you make in a month?'))
```
Challenge

Ask the user for two numbers. You can assume they will be floating point numbers.

Compute the following and print it out to the user:

- The sum of the numbers
- The product of the numbers
- The difference between the numbers
- The first number divided by the second number
Challenge

Write a program that asks the user for a number of pennies, nickels, dimes and quarters

Calculate the total amount of money that the user has and print it out
Challenge

Write a program that asks the user for the value of their current Metro card.

Compute how many rides they have left on their card. Only provide whole number results (i.e. you cannot have 3.5 rides left on a card).
Errors, Bugs and Debugging
"...an analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders."

- Lady Augusta Ada King, Countess of Lovelace (1843)
“It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise —this thing gives out and [it is] then that 'Bugs' — as such little faults and difficulties are called—show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached.”

- Thomas Edison, 1878
Debugging

0800 Aircraft started
0900 Stopped - altitude / 9,000' 
1500 800 mps
1545 Officer p'sd. speed 140 knots TKG

1545 Started Cosine Tape (Sine Check)
1525 Started Multi-Added Test.

1545 Relay 70 Panel F

First actual case of bug being found.
1700 Aircraft started.
1700 Closed down.
Debugging

De-bugging a program is the process of finding and resolving errors.
Types of Errors

We have syntax errors which is code that does not follow the rules of the language.

i.e. We use a single quote where a double quote is needed… A colon is missing… or we use a keyword as a variable name.
We have **runtime errors** which typically involves a program “crashing” or not running as expected.

i.e. You are dividing two numbers but do not test for a zero divisor. This causes a run time error when the program tries to divide by zero.
Types of Errors

We have **logic errors**. These tend to be harder to find and involve code that is syntactically correct, will run but the anticipated result is outright wrong.

i.e. Your program prints “2+2=5”
print("Hello, World! Are you having a fabulous day? I know I am'")
Types of Errors

Source

num = input('Give me a number: ')  
num_float = float(num)  
new_num = 10 + num_float  
print (new_num)

Execution

Give me a number: Eight  
Traceback (most recent call last):  
  File "/Users/HarryPotter/Documents/madlibs01.py", line 6, in <module>  
    new_num = 10 + num  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
Types of Errors

Source

```
num_1 = float(input('give me a num: '))
num_2 = float(input('give me another num: '))
print ('the sum is: ', num_1 - num_2)
```

Execution

```
give me a num: 5
give me another num: 2
the sum is: 3.0
```
Basic Debugging Techniques

Set small, incremental goals for your program. Don’t try and write large programs all at once.

Stop and test your work often as you go. Celebrate small successes.

Use comments to have Python ignore certain lines that are giving you trouble.
Questions???
Python contains two different division operators

The “/” operator is used to calculate the floating-point result of a division operation

The “//” operator is used to calculate the integer result of a division operation (essentially throwing away the remainder). This operation will always round down.

Most times you will use the floating point division operator (“/”)
More Math Operations: Division Operations

```
print (5/2)          # 2.5
print (5//2)         # 2
print (-5/2)        # -2.5
print (-5//2)        # -3
```
Python supports the standard order of operations (PEMDAS)
You can use parenthetical notation inside your math expressions to group operations

Example:

$$((5+10+20)/60) \times 100$$
Write a program that asks the user for three price values.

Calculate the average price in a single variable and output it to the user.
Exponents

You can raise any number to a power by using the "**" operator.

Example:

\[ 2^4 \]

\[ 2 ** 4 \]
Challenge

Calculate the area of a square
Remainder Operator AKA Modulo

The modulo operator (“%”) returns the remainder portion of a division operation.

Example:

5 / 2  # 2.5
5 % 2  # 1
Ask the user to input a number of seconds as a whole number. Then express the time value inputted as a combination of minutes and seconds.

Enter seconds: 110
That’s 1 minute and 50 seconds
Most math formulas need to be converted into a format that Python can understand before they can be evaluated.
Converting Math Formulas into Programming Statements

\[
\begin{align*}
10b & \quad 10 \times b \\
(3)(12) & \quad 3 \times 12 \\
4xy & \quad 4 \times x \times y \\
y = 3\frac{x}{2} & \quad y = 3 \times x / 2
\end{align*}
\]
In this exercise you will ask the user to input the following values

- How much money they want to generate
- An interest rate value
- How long they’d like to invest their money

Calculate how much they will need as an initial investment

Example:

You will need _____ dollars to generate _____ dollars at _____ % over _____ years.
Challenge

\[ P = \frac{F}{(1 + r)^n} \]

P = Present Value
F = Future Value
R = Rate or Return
N = Number of Years
Python allows you to mix ints and floats when performing calculations.

The result of a mixed-type expression will evaluate based on the operands used in the expression.
Mixed Type Expressions

<table>
<thead>
<tr>
<th>operand 1</th>
<th>operand 2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>int</td>
<td>float</td>
<td>float</td>
</tr>
</tbody>
</table>
When using the print() function you probably have noticed that Python automatically places a newline character at the end of each line.

You can override this behavior and have Python use a character of your choice by using the optional ‘end’ argument when using the print() function.

Example:

```python
print ('one', end='')
print ('two', end='')
```
By default, Python will place a space between arguments that you use in print() function calls.

You can override this behavior by using the optional `sep` argument.

Example:

```python
print ('one', 'two', sep='*')
```

# output: one*two
You can use both the ‘sep’ and the ‘end’ arguments at the same time in the same print() function call.

Example:

```python
print ('a', 'b', 'c', sep='*', end='')
```
Most programming languages support an “escape character” that allows you to perform special actions inside the confines of a delimiter.

In Python the escape character is the “\” character

It causes Python to treat the next character as a “special” character – in most cases this means that you will ignore the next character and prevent it from interfering with your delimiter

Example:

```python
print ('Hi, I\'m Cruella de Vil, your professor')
```
Escape Characters

There are a number of special characters you can use in conjunction with the escape character to perform special string operations.

Example – “\n” – forces a line break.

```
print ('line 1\nline 2\nline 3\n')
```

# line 1
# line 2
# line 3
Escape Characters

Example – "\t" – forces a tab:

```python
x = 5
y = 4

print ('X', '\t', 'Y', '\t', 'X*Y')
print (x, '\t', y, '\t', x*y)

X  Y  X*Y
5  4  20
```
Write a program that asks the user to enter in 3 products and 3 prices.

Format your output to look like the following:

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>product1</td>
<td>price1</td>
</tr>
<tr>
<td>product2</td>
<td>price2</td>
</tr>
<tr>
<td>product3</td>
<td>price3</td>
</tr>
</tbody>
</table>
You can’t “add” strings together, but you can “concatenate” them into a single compound string

Example:

```python
a = input('first name')
b = input('last name')
c = b + ',' + a

print (c)
```
You can also “multiply” a string by an integer value to produce a larger string

Example:

```python
lyrics = 'Fa ' + 'La ' * 8
print (lyrics)
```

# Fa La La La La La La La

# Fa La La La La La La La
The format() function can be used to format a string before you decide to print it out to the user.

format() takes two arguments – a number and a formatting pattern (expressed as a string).

format() returns a string which can be treated like any other string (i.e. you can print it out immediately, store its value in a variable, etc).
The first argument passed to the format function is the item that you wish to format.

The second argument passed to the function is the formatting “pattern” you wish to apply to this item.

This pattern varies based on what you would like to do to the item in question.

Once the pattern is applied to the item the format function will return a string version of your item with the formatting pattern applied.
One common use of string formatting is to generate a string that contains a known number of characters.

For example, say you have the strings “Harry” and “Computer Science”. You might want to generate output that looks like the following given these items:

<table>
<thead>
<tr>
<th>Name</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harry</td>
<td>Computer Science</td>
</tr>
</tbody>
</table>

In this case we need to ensure that the strings “Name” and “Harry” are the same width so that the strings that come after them (“Department” and “Computer Science”) line up correctly.
Formatting Strings

You can use the format() function to “pad” a string with extra spaces at the beginning or the end of the string.

For example:

```python
x = format('Harry', '<20s')
```

This will generate a new string (x) that contains the string ‘Harry’ plus 15 spaces at the end of the string. The total length of the string in this case will be 20 characters.

The ‘<‘ character in the formatting pattern tells Python to left justify the string and place the extra spaces at the end of the new string.
You can also have Python right justify the string and place the spaces at the beginning of the new string by doing the following:

```python
b = format('Harry', '>20s')
```
x = "Hello, World!"
y = format(x, '>20s')
print (x)
>>> Hello, World!
print (y)
>>> Hello, World!
The format() function can also be used to generate a printable string version of a float or integer number.

format() takes two arguments – a number and a formatting pattern (expressed as a string).

format() returns a string which can be treated like any other string (i.e. you can print it out immediately, store its value in a variable, etc).
a = 1/6
print (a) 0.16666666666666666

b = format (a, '.2f')
print (b) 0.17
a = 10000/6
b = format (a, ‘.2f’) # format a as a 2 digit float
c = format (a, ‘.5f’) # format a as a 5 digit float
d = format (a, ‘,,.5f’) # 5 digit float + comma separators
e = format (a, ‘>20,.2f’) # 2 digit float, commas + 20 character minimum field width, justified to the right
a = 0.52

print (format(a, '%'))  52.000000%
print (format(a, '.2%'))  52.00%
print (format(a, '.0%'))  52%
a = 20000

print (format(a, ',d'))  20,000
print (format(a, '>20,d'))  20,000
Write a program that generates the 2 times table, like this:

<table>
<thead>
<tr>
<th>Number 1</th>
<th>Number 2</th>
<th>N1 * N</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
next steps:

begin “Self Paced Learning Module # 3”

work on Assignment #2: ‘Input, Processing, Output!’