Python for FLL Teams: Core

MD FLL Mentors Group
7 September 2025
Rev 26

Boyd Fletcher
fllcoachboyd@gmail.com

Why Python?

- It will help you progress in the STEM program, especially in high school
- It is a real programming language and used widely in industry, science, engineering, and mathematics
- It is more flexible than Scratch (graphical/block programming)
- Compared to visual programming like Scratch:
    - It is easier to understand what your code is doing
    - It is easier to troubleshoot your code and find errors (called 'debugging')
    - It is easier to share your code with others
    - Code reuse is much easier
    - Revision (version) control is MUCH easier
    - There thousands of software libraries you can use that other people have written

Sources for Slides

These slides are heavily derived from multiple sources including presentations and documentation from:

- UMBC's python course
- Forschungszentrum Jülich
- Marty Stepp
- Moshe Goldstein
- Tao Yang at UCSB
- Bernard Chen, University of Central Arkansas
- Lego Spike™ documentation
- python.org official documentation
- pybricks.com – the official site for Pybricks

A Quick Introduction to Programming Concepts

What do I use to create a Program? (1/2)

- Programs can be written in any application that can create plain text files
    - NotePad/TextEdit, WordPad, emacs, vi, nano, Visual Studio Code
- However, most programs are written in special purpose software called an Integrated Development Environment (IDE)
- IDE’s contain a program called an “Editor” that lets you create the program
- IDE’s understand the program language and can provide ‘hints’ when you are writing to the code to help with how the language works

What do I use to create a Program? (2/2)

- IDEs can either compile the program you are writing or execute it in an interpreter
- Examples of IDEs include Lego Spike Prime, Pybricks Code, NetBeans, Eclipse, Visual Studio, Xcode
- Most IDEs append a file extension to the end of the file to signify, the programming language used:
  - `.py` is used for python programs
  - `.java` is used for Java, `.c` is used for C, `.cpp` for C++, `.js` is used for JavaScript

For FLL, the Python we create will be in the Pybricks Web App that is running in the Chrome Web Browser

Compiled vs. Interpreted Programs (1/2)

Compiled programs are created in an editor and compiled using a compiler into an executable (e.g., a binary, a .exe file)

- Most programs running on a PC, Mac, iPhone, iPad, Switch, Xbox, PlayStation are compiled
- Programming languages like C, C++, C#, Java, and Rust are compiled
- Compiled programs are also called binaries
- Compiled programs cannot be edited in an IDE

Compiled vs. Interpreted Programs (2/2)

Interpreted programs are created in an editor and but are run (e.g., executed) inside of the interpreter

- Easy to make changes
- Programming languages like Python, Perl, PowerShell, JavaScript, and Scratch
  - A Web Browser (e.g., Chrome) has an interpreter for JavaScript
- Interpreted languages are generally faster to develop with but slower in performance
- If you share an interpreted language-based program, people can see how you did it and change it

For FLL, the Python programs we create will be interpreted by the Pybrick's MicroPython interpreter running on the robot!

Version/Revision Control (1/2)

When creating your program, you will:

- Have new ideas to try: some will work and some will not
- Add more capabilities to your programs over time
- Make mistakes implementing those new ideas and capabilities
- Want to "return" to a previously working version
- If you always use the same filename for your code, you can't go back in time to a previous version

Version/Revision Control (2/2)

- If you save files with a date or/and revision number (#) added to the file name you can go back in time to a previous version
  - For example, instead of using "`myprogram.py`" as the file name, try "`myprogram_rev1.py`" and change the rev # each time you make updates to the file

- In large software projects, source code control systems provide a better way to revert back in time without changing filenames
  - The most popular source code control system is called **git**

For FLL teams just starting with Python we recommend using the _rev# approach

Debugging your Program

- When software has a mistake its called a ’bug’
- The activity to find and fix bugs is called ‘debugging’
- All reasonably complex programs have bugs, especially early in their development and testing
- Some IDE’s have special capabilities to support debugging your code
- We will discuss debugging techniques later in the course

Programming Language vs. Application Programming Interface (API) (1/2)

- All programming languages have a set of commands and functions that make up the core language
- They include the ability to store/modify data, create and run loops, and make decisions
- However in order for a program to interact with the operating system of the computer, the user of a program, a Lego hub, Lego motors, and Lego sensors special software called **Application Programming Interfaces (API)** have been created

Programming Language vs. Application Programming Interface (API) (2/2)

- APIs are sets of functions that perform specific jobs or tasks
  - A **function** is a reusable set of code
  - Examples include: reading/writing to files, drawing lines or rectangles on a screen, controlling a motor, checking a sensor
- A grouping of API functions is called a **library**
- Anyone can create their own APIs and libraries

For FLL, the we will be using a combination of the Pybricks API and developing our Robot Library API

Introduction to Python

Disclaimer

- This section of the presentation is about learning Python3 as a programming language
- This is presentation does **not** cover all the capabilities of Python
- It is primarily focused on the Python language features most useful for beginner and intermediate Lego Robotics
- To experiment with Python3 on your own computer use Visual Studio Code not Lego Spike Prime or Pybricks
- Not all features of Python3 are available in the Lego Spike or Pybricks versions of MicroPython

References

- Authoritative Python Site:
  - https://www.python.org
- Python 3 Cheat Sheet (great 2 page reference!)
  - https://scouv.lisn.upsaclay.fr/python-memento/index.en.html
- Microsoft Visual Studio Code
  - https://code.visualstudio.com
- Pybricks Documentation
  - https://pybricks.com
- MicroPython (used by both Lego Spike & Pybricks)
  - https://micropython.org
- TutorialsPoint
  - https://www.tutorialspoint.com/python/index.htm
- W3 Schools
  - This site includes documentation on Python and a way to run your Python code via the Web
  - https://www.w3schools.com/python/default.asp

Download and Print this Cheat Sheet

This is my favorite!

How to ”read” the slides

- Explanations of concepts and ideas will be in this font (called Calibri) and in black
- Examples of code will be in **`Courier New`** font, with
  - **`Black text`** for Variables and the Python Core Language
  - **`Blue text`** for Python Standard Functions and Pybricks Functions
  - **`Red text`** for Strings
  - **`Green text`** for Numbers, Booleans, Comments
  - Source code examples will usually be in a light grey box .
  - The output of running a program will be in white text in a black box
- Examples with **>>>** means this examples were entered directly into the Python Interpreter
- A newline is created by pressing the ENTER or RETURN key

Creating and Running the Example Code

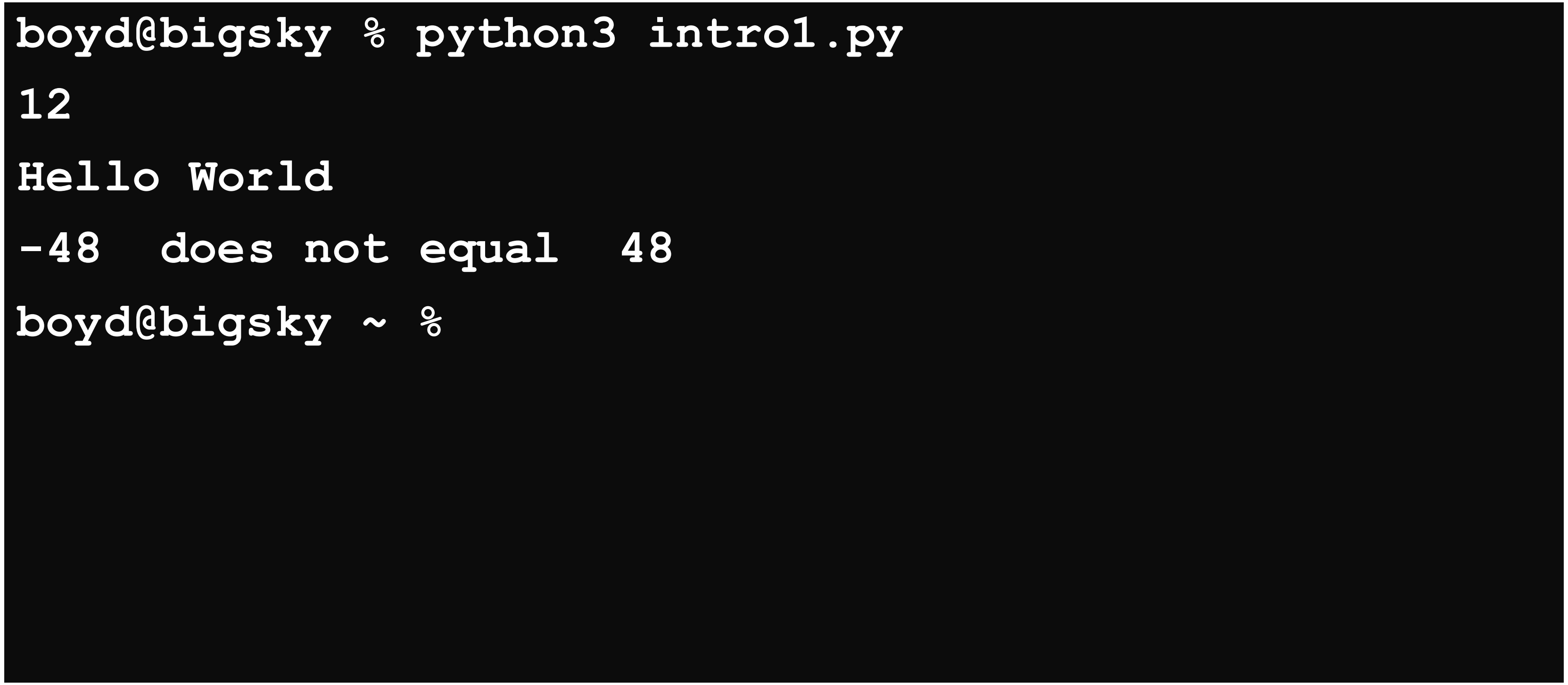
- The examples in the course assume you have installed Visual Studio Code (VS Code), Python 3, Chrome, and Pybricks installed on your computer
- The early examples will focus on using VS Code with Python3
- Robot examples will use the Pybricks Code application
- The examples in the course were created and run on Mac; however a Windows or Linux based system would be very similar

A Code Example (intro1.py)

- Start Visual Studio Code
- Enter the below program
    - You don’t need to enter the comments (everything after the # in green) for the program to work.
- And save it as “`intro1.py`”

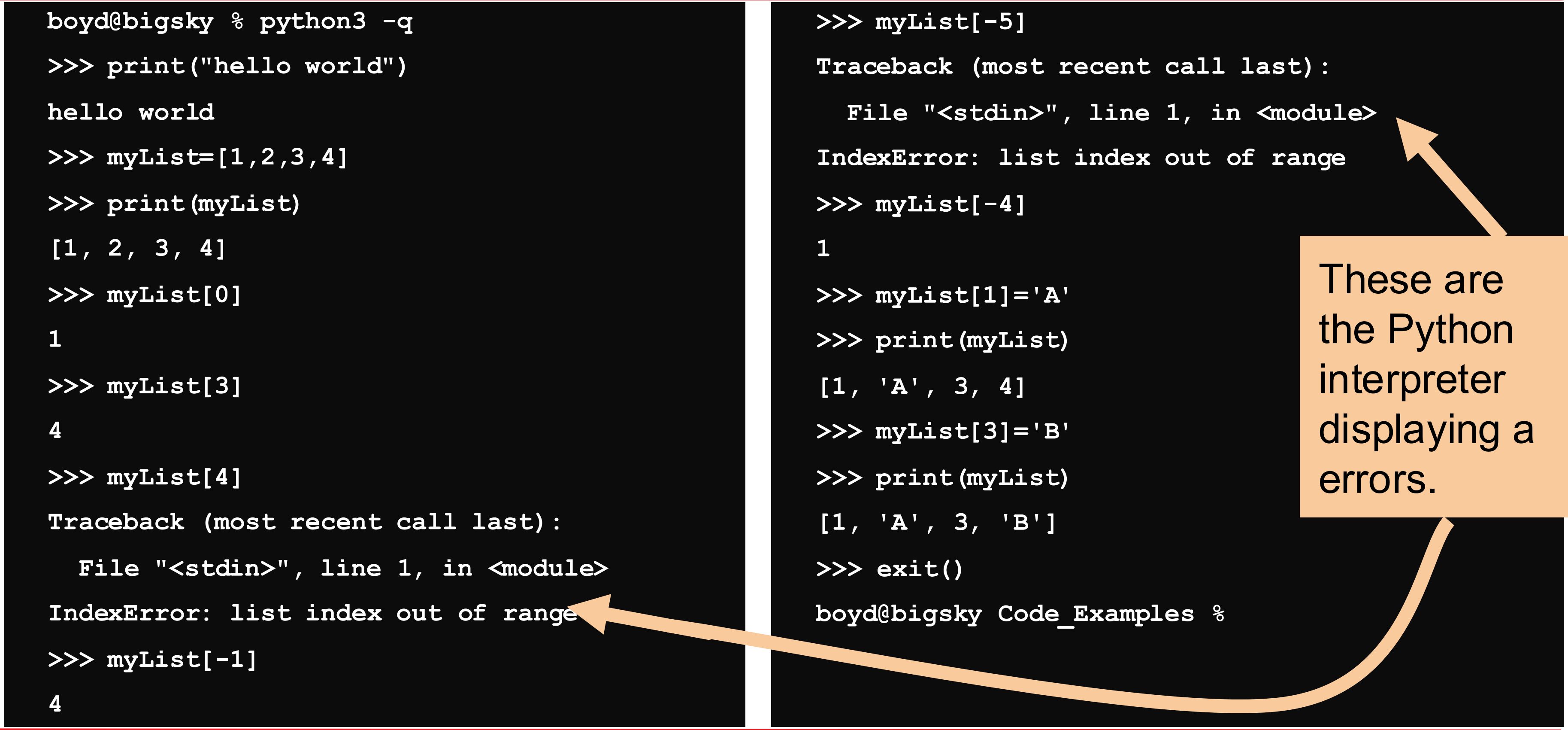
```
x = 34 - 23                     # A comment about setting the variable x to the
                       # value of 34 minus 23
y = "Hello"                     # A comment about setting the variable y to “Hello”
z = (x * -5) + 7
if (z <= -5 and y == "Hello"):
         x = x + 1
         y = y + " World"          # Concatenating (combining) two strings
print(x)
print(y)
print(z," does not equal ",abs(z))     # abs() is a function that returns the
                                       # absolute value of a number
```

A Code Example: Running the Program



```
boyd@bigsky % python3 intro1.py
12
Hello World
-48  does not equal  48
boyd@bigsky ~ %
```

Another Code Example but using the Python Interpreter



```
boyd@bigsky % python3 -q
>>> print("hello world")
hello world
>>> myList=[1,2,3,4]
>>> print(myList)
[1, 2, 3, 4]
>>> myList[0]
1
>>> myList[3]
4
>>> myList[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> myList[-1]
4
```

```
>>> myList[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> myList[-4]
1
>>> myList[1]='A'
>>> print(myList)
[1, 'A', 3, 4]
>>> myList[3]='B'
>>> print(myList)
[1, 'A', 3, 'B']
>>> exit()
boyd@bigsky Code_Examples %
```

These are the Python interpreter displaying a errors.

Python Language “Rules” 1/2 (abbreviated)

- Python is case sensitive including variable names and function names
  - Variable **A** and variable **a** are not the same variable
  - Function `DriveForward()` is not the same as function `driveforward()`
- Function names and variables cannot start with a number
  - They can contain letters, numbers, and underscores.

```
bob   Bob   _bob   _2_bob_   bob_2   BoB
```

- Python starts counting at position 0 for Lists and most other data types not 1
- Whitespace (spaces & tabs) is meaningful in Python: especially indentation and placement of newlines
  - Use a newline (by pressing ENTER or RETURN) to end a line of code
  - Use \ at the end of the line to continue code onto next line

Python Language "Rules" 2/2 (abbreviated)

- Colons start a new block of code in many language constructs (e.g., `if`, `while`, `for`)
- The basic printing command is `print`
- There are some reserved words:

```
and, assert, async, await, break, class, continue, def,
del, elif, else, except, exec, False, finally, for, from,
global, if, import, in, is, lambda, None, not, or, pass,
print, raise, return, True, try, while
```

Camels and Underscores

- The “official” python way to separate words in Variable and Function names is to use the underscore (_)
  - Examples:
    - **`Num_Of_Iterations`**
    - **`Forward_Velocity`**
    - **`Drive_Robot_Forward()`**

- Alternatively, many other programming languages like C, C++, and Java use CamelCase for variable and function names
  - First letter of each word is Capitalized.
    - Variant: Some programmers make the first word all lowercase, but rest of the words have first character capitalized
  - Examples:
    - **`numOfIterations`**
    - **`ForwardVelocity`**
    - **`DriveRobotForward()`**

- Either way works. Just pick one for the team and be consistent!
  - I will use CamelCase because I grew up using programming in the C programming language

Identifying Blocks of Code

- Python uses consistent indentation to identify blocks of code
    - Languages like C/C++/C#, JavaScript, and Java use **{ }** instead
- The first line with less indentation is outside of the block (see For, If, While examples)
- The first line with more indentation starts a nested block (see For, If, While examples)
- Tabs or Spaces can be used. Space are more reliable (3-4 spaces are normal)
    - Lego Spike Prime converts tabs to spaces (usually 5) – sorta but treats them as tab when backspacing/deleting them. Odd behavior.
    - Pybricks seems to handle them correctly
    - **Just don't use tabs**

`print` Function

- `print` : Produces text output on the console.
  - Syntax:

```
print("Message")

print(Expression)

print(Item1, Item2, ..., ItemN)
```

- Prints the given text message or expression value on the screen, and moves the cursor down to the next line.
- The `print` function is the major tool in Python and Lego Spike Prime in helping to debug your code!
- Adding an `end='characters'` to the `print` command can be used to prevent printing a newline. Basically, if you do `end=' '` it will print a space instead of moving to next line (see `For` command examples).

```
name="John Smith"
age=42
print("The Patient's name and age is ",name,",",age)
```

```
boyd@bigsky ~ % python3 print.py
The Patient's name and age is  John Smith, 42
boyd@bigsky ~ %
```

Comments

- Comments are used to document what your code is intended to do
- It is a reminder to you and a tool to help others learn!
- Start comments with #, rest of the line is ignored after the #
  - If you want to disable a command, it is best to put the # as the first character of the line.
- Multiline comments start and end with `'''`

```
'''
this is a multiline comment
about comments
'''
```

With Micro Python (what is used on the Robots) the `'''` must be at the **same** indent level as the block code you are commenting out!

Data Types & Variables

Storing your data

What is a Variable?

- A **variable** is mechanism to **store data** in a program
- Each variable has a data type
- First assignment to a variable creates it
- **Variables are assigned names** so that they can be used
- In Python, the **names are case sensitive** which means `MyVariable` and `myvariable` are different variable names
- Not all data types can be changed once created

Example Data Types

- Integers
  - `1, -5, 2000, 5837293, -768`
- Floats (also called Floating Point numbers or Decimal numbers)
  - `1.5, 3.14, -94.2383`
- Strings
  - `“cat” , “my house is red”`
- Boolean
  - `True, False`

Miscellaneous Variable Facts

- The single equal sign (**=**) is used to assign a value to variable
- The double equal sign (==) is used to compare two values (see **`if`** statements in the program flow section)
- The plus sign (**+**) when used with strings is used for concatenation (combining strings together)
- Data type of a variable is determined the first time you assign data to the variable
- Add Array examples and type casting

Data Typing

- **Dynamic typing***:* Python determines the data types of ***variables*** in a program automatically
- **Strong typing:** But Python’s not casual about types, it enforces the types of variables
    - This means you can’t mix types unless you convert them to a common type
    - For example, you can’t just append an integer to a string, it must first be converted to a string using the **`str()`** function

```
>>> x = "the answer is "     # x bound to a string
>>> y = 23                   # y bound to an integer
>>> print(x + y)             # Python will complain!
Traceback (most recent call last):
  File "<python-input-38>", line 1, in <module>
    print(x + y)             # Python will complain!
          ~~^~~
TypeError: can only concatenate str (not "int") to str
>>>
>>>
>>> x = "the answer is "     # x bound to a string
>>> y = 23                   # y bound to an integer
>>> print(x + str(y))        # this is ok, y is converted to a string
the answer is 23
>>>
```

This is the Python interpreter displaying an error.

Global vs. Local Variables

- Variables created in a function are local to the function
  - This means they do not exist outside the function
- Variables created outside of a function are global (e.g., available to the whole program)
- Normally, if you change a global variable in a function, the change does not leave the function
- If you want to use or change a global variable in a function use the `global` keyword in front the variable name when you assign a new value to it.
- Some good examples are at: https://www.w3schools.com/python/python_variables_global.asp

Local vs. Global Variable Example

```python
ExampleGlobal="i'm global"

def LocalVariableTest1():
    ExampleLocal="i'm local"
    ExampleGlobal="i'm now local too"
    print("--- LocalVariableTest1 Function --- start")
    print(ExampleLocal)
    print(ExampleGlobal)
    print("--- LocalVariableTest1 Function --- end ")

def LocalVariableTest2():
    ExampleLocal="i'm local"
    global ExampleGlobal
    ExampleGlobal="i'm a changed global"
    print("--- LocalVariableTest2 Function --- start")
    print(ExampleLocal)
    print(ExampleGlobal)
    print("--- LocalVariableTest2 Function --- end")

print(ExampleGlobal)
LocalVariableTest1()
print(ExampleGlobal) # notice how the value didn't change
LocalVariableTest2()
print(ExampleGlobal) # notice how the value did change
```

```
boyd@mac Code_Examples % python3 global_vs_local_demo.py
i'm global
-— LocalVariableTest1 Function -— start
i'm local
i'm now local too
-— LocalVariableTest1 Function -— end
i'm global
-— LocalVariableTest2 Function -— start
i'm local
i'm a changed global
-— LocalVariableTest2 Function -— end
i'm a changed global
boyd@mac Code_Examples %
```

Math in Python

Calculate This!

Mathematical Expressions

- **Expression**: A data value or set of operations to compute a value.

```
1 + 4 * 3
```

- **Arithmetic operators** are used to create mathematical expressions

`**`	exponentiation
`* /`	multiplication, division
`+ -`	addition, subtraction/negation
`%`	modulus, a.k.a. remainder

- **Precedence**: Order in which operations are computed.

  `* / % **` have a higher precedence than `+ -`

```
 1 + 3 * 4 = 13
```

- **Parentheses** can be used to force a certain order of evaluation.

```
(1 + 3) * 4 = 16
```

Order of Operations (also called precedence)

- The rules that are used to evaluate mathematical expressions (e.g., equations)
- Programming Languages implement order of operations
- Python uses PEMDAS:
  - **P**arentheses, **E**xponents, **M**ultiplication and **D**ivision (from left to right), **A**ddition and **S**ubtraction (from left to right)

Other Functions on Numbers

- Absolute value: **`abs(x)`**
- Rounding: **`round(x)`**
- Conversion: **`int(x), float(x)`**
- Power: **`pow(x, y)`**
- Python's built-in functions are:
  - https://docs.python.org/3/library/functions.html
- Python also has an extensive math API of even more mathematical functions at:
  - https://docs.python.org/3/library/numeric.html

Some Math Examples

```
boyd@bigsky ~ % python3
>>>
>>> x = 5
>>> y = 6
>>> z = -8
>>> print(x,y,z)
5 6 -8
>>> print(abs(z))
8
>>> a=3.333333
>>> b=-456.789
>>> print(abs(a), abs(b))
3.333333 456.789
```

```
>>> print(int(a))
3
>>> print(int(b))
-456
>>> print(float(x))
5.0
>>> print(pow(x,y))
15625
>>> print(x**y)
15625
>>> c=round(a)
>>> print(c)
3
```

Boolean Logic Expressions

Are you being "Truthful"?

Boolean Logic Expressions

- Boolean Logic Expressions are the basis for how computers make decisions
- All programming language commands used to make decisions are based on expressions that result in either **True** or **False**
- **`if`**, **`while`**, and **`for`** statements (discussed later) all use Boolean Expressions to determine what to do
- Mathematical expressions that result in a **True** or **False** are commonly used used in `if`, `for`, and `while` commands

Boolean Values

- Data type **bool:** `True, False`
- Some variable values that are evaluated to `False`
  - **`None`**
  - False
  - 0
  - Empty string, lists, and tuples: ‘’, [], ()
  - Empty dictionaries: {}
  - Empty sets: set()
- All other objects of built-in data types are evaluated to `True` including non-zero numbers, non-empty objects

Boolean Truth Table

- `a and b` *True* if **a** is **True** and **b** is **True**
- `a or b` *True* if **a** is **True** or **b** is **True**:
- `not a` *True* if **a** is False:

Boolean Truth Table				
A	**B**	**A and B**	**A or B**	**not A**
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

- Use parentheses as needed to disambiguate complex Boolean expressions.
- Comparison operators: `==, !=, <, <=, >, >=` results in a `True` or `False`
- To determine if X and Y have same value use: `X == Y`
- To determine if X and Y are the same object use: `X is Y`

Boolean Logic Expression Examples

```python
A=True
B=False
C=True
print("A=", A, " B=",B," C=",C)
print("A and B = ", A and B)
print("A and C = ", A and C)
print("A or B = ", A or B)
print("A or C = ",A or C)
J=10
K=100
L=100/10
print("J=", J, " K=",K," L=",L)
print("J < K =", J < K)
print("K > J =", K > J)
print("J == K =", J == K)
print("J == L =",J == L)
```

```
boyd@bigsky ~ % python3 boolean.py
A= True  B= False  C= True
A and B =  False
A and C =  True
A or B =  True
A or C =  True
J= 10  K= 100  L= 10.0
J < K = True
K > J = True
J == K = False
J == L = True
boyd@bigsky ~ %
```

Program Flow Control

Or how to make a program do something useful

The Infamous “If..Then” Statement

- All programming languages have the ability for programmers to ask a question in code and then take one or more actions based on the answer

- The answer to the questions always results in either a **True** or **False** answer

- In software, the “if..then” statements are also called branching and allows a program to take different actions based of the result of mathematical or logical expressions

- Most programming languages can stack (also called cascade) “if..then” statements

`if/elif/else` Statements

- `if/elif/else` **statement**:
  - An `if` statement executes a group of statements only if a certain condition is **True**. Otherwise, the statements are skipped.
  - An `if/elif/else` series of statements executes ***statements1*** if the first condition is **True**, executes ***statements2*** if the second condition is **True** and the first condition is **False**, and executes ***statements3*** if the first and second conditions are **False**.

Syntax:

```
if condition:
          statements1
elif condition:
          statements2
else:
          statements3
```

Reminder!

- To indent blocks of statements
- Put a Colon (:) after the condition (Boolean expression)

- Multiple conditions can be chained with `elif` ("else if")
- `elif` and `else` are optional
  - You can only have one `else` but an unlimited number of `elif` statements

`if/elif/else` Examples

```
A = 10
B = 15
if (A > B):
   print("A > B")

if (A == B):
    print("A = B")
elif (A < B):
    print("A < B")

if (A == B):
   print("A = B")
elif (A > B):
   print("A > B")
else:
   print("B > A")
```

```
boyd@bigsky ~ % python3 if_elif_else.py
A < B
B > A
boyd@bigsky ~ %
```

`for` Loop

- `for` loop: Repeats a set of statements over a group of values

  Syntax:

  ```
  for variableName in groupOfValues:
          statements
  ```

  - Indent the statements to be repeated
  - ***variableName*** gives a name to each value, so you can refer to it in the ***statements***.
  - ***groupOfValues*** can be a range of integers, specified with the **range** function, or it can also be a list, string, or tuple

`range` Function

- The `range` function specifies a range of **integers**:
- `range(`*`start`*`, `*`stop`*`, `*`step`*`)`
    - *`start`* is optional, inclusive, and defaults to 0
    - *`stop`* is required but exclusive
    - *`step`* is optional and defaults to 1
- ***Inclusive*** means that number will be part of the range. So if you start at "1" then 1 will be the first number
- ***Exclusive*** means the number will **not** be part of the range. So if use the **range(0,100)** the ending (**stop**) number will be 99 not 100.

for Loop Example

```
myStep=-10
startCounter=100
endCounter=1
for x in range(startCounter, endCounter, myStep):
    print(x,' ',end='') # note the use of end='' to tell print not to add a newline
print()
```

```
boyd@bigsky ~ % /usr/local/bin/python3 for.py
100  90  80  70  60  50  40  30  20  10
boyd@bigsky ~ %
```

```
myValues=['a', 'b', 'c', 'd', 'e']
for x in myValues:
    print(x,end=':')
print()
```

```
boyd@bigsky ~ % /usr/local/bin/python3 for2.py
a:b:c:d:e:
boyd@bigsky ~ %
```

`while` Loops

- **`while` loop**: Executes a group of statements as long as a condition is True.
  - good for *indefinite loops* (repeat an unknown number of times)

Syntax:

`while` ***condition*:**

```
number = 1
while (number < 200):
    print(number,' ',end='')
    number = number * 2
print() # print just a newline
```

```
boyd@bigsky ~ % /usr/local/bin/python3 /Users/boyd/Documents/Python/Code_Examples/while.py
1  2  4  8  16  32  64  128
boyd@bigsky ~ %
```

`break`, **`continue`**, and **`pass`**

- The keyword **`break`** is used inside a loop to leave the loop
  - Works in **`for`** and **`while`** loops

- The keyword **`continue`** is used inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one (basically skips
  - Works in **`for`** and **`while`** loops

- The keyword **`pass`** is used inside a loop or if/elif/else statements to do nothing. Basically it skips to the next sequential statement.

break, continue, and pass examples

```
print("break demo")
for i in range(1,10,1):
if i > 5:
  break
print(i)
print("continue demo")
for i in range(1,10,1):
if i % 2 == 0:
  continue
print(i)
print("pass demo")
i=0
while i < 10:
if i % 2 != 0:
  pass
else:
  print (i)
i = i + 1
print("end of demo")
```

```
break demo
1
2
3
4
5
continue demo
1
3
5
7
9
pass demo
0
2
4
6
8
end of demo
```

Functions (aka Methods)

Code reuse is key

Functions

- A way to group related source code so it can be reused
- Functions are “called” (“executed”) from other parts of your program
- Functions can call other functions
- You can, optionally, pass variables (called arguments) into a function so that it can use the data stored in the arguments to perform work
- Using functions is good programming practice
- Functions enable your programs to be modular which makes debugging easier
- In some programming languages functions are called methods or procedures

Defining Functions

No declaration of <u>types</u> of function or arguments is required.

Function definition begins with “def.”

Function name and its arguments.

```
def get_final_answer(a,b):
    “““Documentation String”””
    print(a)
    print(b)
    return (a*b/3.14)
```

Co

lon

The indentation matters…
The first line with less indentation is considered to be outside of the function definition.

The keyword ‘**return**’ indicates the value to be sent back to the caller of the function.

Functions and Return Values

- Functions can optionally return a value (e.g., the result of a calculation or whether an action was successful)
- Use the **`return`** command to return a value from a function
- *All* functions in Python have a return value, even if no **return** line is present
- Functions without a **`return`** return the special value **`None`**
    - **`None`** is a special constant in the language
    - **`None`** is used like *NULL*, *void*, or *nil* in other languages
    - **`None`** is also logically equivalent to **False**

Variables and Functions

- Variables defined in a function are local to the function and can only be used in the function

- Variables defined outside of functions are called global variables and can be used within the function

- If a global variable is changed inside a function the change stays local to the function
    - Unless the word **`global`** is used to mark the variable as a global variable in the function

Variables and Functions

```
temp=29

def WearJacketToday(temp):
  print('temperature in func Jacket=',temp)
  if (temp < 40):
     return True
  return False

def WearHatToday():
  global temp
  temp = -5
  print('temperature in func Hat=',temp)
  if (temp < 40):
     return True
  return False

print('temperature before func=',temp)
print("Wear a Jacket Today? ",WearJacketToday(80))
print('temperature after func=',temp)
print("Wear a Hat Today?",WearHatToday())
print('temperature after func=',temp)
```

```
boyd@bigsky ~ % /usr/local/bin/python3 var_test.py
temperature before func= 29
temperature in func Jacket= 80
Wear a Jacket Today?  False
temperature after func= 29
temperature in func Hat= -5
Wear a Hat Today? True
temperature after func= -5
boyd@bigsky ~ %
```

Importing Modules, Functions, & Variables

How to package and share your code

Importing Modules

- Many programming languages group related functions, classes, and objects into a module
- You can use modules to break up your program into individual files
- If you want to use functions and variables/objects from a module, you have explicitly request them using the **`from`** and **`import`** commands

```
import [module]                                # imports an entire module
from [module] import [function or variable]    # imports specific functions or variables from a module, separated by commas
```

- The **`as`** keyword can be used to override the default name of the module

```
import math          # import the math module, but you have to put the module name and a period (math. In this example)
                     # in front of all functions from that module
s = math.sin (math.pi)
```

```
import math as m              # import the math module, but assign a name of m
s = m.sin (m.pi )
```

```
from math import pi as PI , sin
s = sin ( PI )
```

```
from math import *                # import all the functions, object, and classes from math
s = sin ( pi )
```

Creating a Module

- Create one or more functions and save them in a file (e.g., mymodule.py)
- Breaking up a program into multiple files is good software engineering
- Create a new program and use the import command load the module you just created

Redo with better examples

myPrintFunctions.py

```
# my print functions

def PrettyPrint(message):
print("<***>",message,"<***>")

def SimplePrint(message):
print("==", message,"==")
```

```
nypf

etty")

mypf.SimplePrint("this is plain")
```

```
boyd@bigsky ~ % /usr/local/bin/python3 myPrintImport.py
<***> isn't this pretty <***>
== this is plain ==
boyd@bigsky ~ %
```

Best Practice:

Breaking your program up into modules improves software maintainability and allows different developers to work on the same project at the same time!

Debugging Code

What is Debugging?

- Testing is the process of finding bugs in your code
- Debugging is the process of identifying where the bugs are in your code and fixing them
- A bug is a flaw in your code:
    - It can be as simple as a typo that causes the code not to execute/run or fail during execution
    - It can be logic flaw
    - It can be a mistake an expression used in **`if/elif/else`** statements, **`for`** statements, or **`while`** statements
    - And many other things…

Tricks to help with Debugging

- Put comments in your code
  - Comments is how you document what your code is doing
  - The more robust your comments, the better the chance you (or some else on your team) will understand the intent behind what the code is supposed to do - even if it is not working correctly
- Use **`print`** statements in your code print where you are in your program
- Use **`print`** statements to print out the content of variables
- Use "`if debug: print("`*`your message here`*`")`" statements as a way to turn on or off debug statements by simply setting the variable **`debug`** to **`True`** (on) or **`False`** (off)
  - NOTE: **`debug`** is this just a variable name, you called it anything you want.
- Some developers will have multiple debug "levels" so they can increase the amount of information displayed

Debugging Example Part 1

```python
# debug example 1
print("**** debug example 1 ****")
# this program has one debug level
debugLevel1=True
# initialize the counter and sum variables
counter=0
sum=0
print("This program prints the summation
of all integers between 1 and 10")
while counter != 10:
    counter = counter + 1
    sum=sum+1
print("Sum equals=",sum)
```

```
boyd@mac Code_Examples % python3 debug_level_1.py
**** debug example 1 ****
This program prints the summation of all integers between 1
and 10
Sum equals= 10
boyd@mac Code_Examples %
```

Debugging Example Part 2

```python
# debug example 2
print("**** debug example 2 ****")
# this program has one debug level
debugLevel1=True
# initialize the counter and sum variables
counter=0
sum=0
print("This program prints the summation of all integers
between 1 and 10")
if debugLevel1: print("start counter=",counter)
while counter != 10:
    counter = counter + 1
    sum=sum+1
    if debugLevel1: print("counter=",counter," sum=",sum)

if debugLevel1: print("\nfinal counter=",counter)
print("Sum equals=",sum)
```

```
boyd@mac Code_Examples % python3 debug_example.py
**** debug example 2 ****
This program prints the summation of all integers
between 1 and 10
start counter= 0
counter= 1   sum= 1
counter= 2   sum= 2
counter= 3   sum= 3
counter= 4   sum= 4
counter= 5   sum= 5
counter= 6   sum= 6
counter= 7   sum= 7
counter= 8   sum= 8
counter= 9   sum= 9
counter= 10   sum= 10

final counter= 10
Sum equals= 10
boyd@mac Code_Examples %
```

Debugging Example Part 3

```python
# debug example 3
print("**** debug example 3 ****")
# this program has one debug level
debugLevel1=True
# initialize the counter and sum variables
counter=0
sum=0
print("This program prints the summation of all integers
between 1 and 10")
if debugLevel1: print("start counter=",counter)
while counter != 10:
   counter = counter + 1
   sum=sum+counter
   if debugLevel1: print("counter=",counter," sum=",sum)

if debugLevel1: print("\nfinal counter=",counter)
print("Sum equals=",sum)
```

```
boyd@mac Code_Examples % python3
debug_example.py
**** debug example 3 ****
This program print the summation of all integers
between 1 and 10
start counter= 0
counter= 1  sum= 1
counter= 2  sum= 3
counter= 3  sum= 6
counter= 4  sum= 10
counter= 5  sum= 15
counter= 6  sum= 21
counter= 7  sum= 28
counter= 8  sum= 36
counter= 9  sum= 45
counter= 10  sum= 55

final counter= 10
Sum equals= 55
boyd@mac Code_Examples %
```