

Introduction to Python Programming

<https://raspberrypicamp.rocks>

General Information

- Unlike C/C++ or Java, Python statements do not end in a semicolon
- In Python, indentation is the way you indicate the scope of a conditional, function, etc.
- Look, no braces!
- Python is interpretive, meaning you don't have to write programs.
- You can just enter statements into the Python environment and they'll execute
- For the most part, we'll be writing programs

The Python Shell

- Because Python is interpretive, you can do simple things with the shell
- In the graphical shell on Linux, double-click on Terminal
- Start by creating a python directory in your home directory
`mkdir ~/python` ↵
- After making the directory, type `Python` ↵
- You should have a `>>>` prompt
- Type in:
`print("hello, world")` ↵
- You have written your first Python program! 👍
- Exit the shell by typing `exit()` ↵

Idle3: The Python IDE

- The shell is good for simple calculations but not for real programming
- For programming, we'll use **Idle3**
- There are two versions: Idle for Python 2 and Idle3 for Python 3...for most of what we do, we'll use Python 3
- Idle will give you access to a **shell** (where each statement executes as soon as you type it and press ↵) but also to an **IDE** for writing and saving programs

Python Modules

- In practice, only the simplest programs are run in the shell
- You can create a module (complete program) by going to the **File->New File** menu option
- This brings up a text editor that lets you create a Python program and run it
- Write your first "Hello World!" program thus:

```
print("Hello, World!")
```

Python Modules

- Press **F5** (or **Run -> Run Module** from the menu)
- It will ask you to save the file before you run it
- Save it to your **python** directory as **HelloWorld.py**
- If you don't provide the **.py** extension, IDLE3 will automatically do it
- If you want to run it outside of the development environment, go into your terminal window and simply type:

```
python HelloWorld.py ↵
```

- **Note that Linux is case sensitive!**

Variables

- In every computer language, a variable is the name of a memory location
- Python is **weakly typed** (that means it tries to figure out what kind of information a variable contains)
- You can store numbers, text (called *strings*) and lots of more complicated things in a variable.
- Variable names begin with a *letter* or an *underscore* and can contain letters, numbers, and underscores
- Python has *reserved words* that you can't use as variable names. You can think of reserved words as predefined words in the Python language...if you used a variable with the same name as a predefined word in its language, Python would get confused!

Variables

- At the `>>>` prompt, do the following:

```
x=5 ↵
```

```
type(x) ↵
```

```
x="this is text" ↵
```

```
type(x) ↵
```


```
x=5.0 ↵
```

```
type(x) ↵
```


Printing

- You've already seen the `print` statement
- You can print numbers with your own customized formatting (how many decimal places, for example) or with default formatting

Comments

- All your code should contain **comments** that describe what it does
- In Python, lines beginning with a **#** sign are comment lines
- On American English keyboards, this is over the  key

You can also have comments on the same line as a statement

```
# This entire line is a comment  
dozen = 12          # 12 in a dozen!
```

Operators

- Arithmetic operators we will use:

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

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

- $*$ $/$ $\%$ $**$ have a higher precedence than $+$ $-$

$1+3*4$ is 13

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

$(1+3)*4$ is 16

Expressions

- When integers and reals are mixed, the result is a real number.
 - Example: $1/2.0$ is 0.5
 - The conversion occurs on a per-operator basis.

$$\begin{array}{r} 7 / 3 * 1.2 + 3 / 2 \\ \underline{2} * 1.2 + 3 / 2 \\ 2.4 + \underline{3 / 2} \\ 2.4 + \underline{1} \\ 3.4 \end{array}$$

- What do you notice above, when an integer is divided by an integer?

ANS: If there's a remainder the result is truncated (or rounded down)

Math Functions

- Use this at the top of your program: **from math import ***

Command name	Description
<code>abs(<i>value</i>)</code>	absolute value
<code>ceil(<i>value</i>)</code>	rounds up
<code>cos(<i>value</i>)</code>	cosine, in radians
<code>floor(<i>value</i>)</code>	rounds down
<code>log(<i>value</i>)</code>	logarithm, base e
<code>log10(<i>value</i>)</code>	logarithm, base 10
<code>max(<i>value1</i>, <i>value2</i>)</code>	larger of two values
<code>min(<i>value1</i>, <i>value2</i>)</code>	smaller of two values
<code>round(<i>value</i>)</code>	nearest whole number
<code>sin(<i>value</i>)</code>	sine, in radians
<code>sqrt(<i>value</i>)</code>	square root

Constant	Description
e	2.7182818...
pi	3.1415926...

Relational Operators

- Many logical expressions use *relational operators*:

Operator	Meaning	Example	Result
==	equals	<code>1 + 1 == 2</code>	True
!=	does not equal	<code>3.2 != 2.5</code>	True
<	less than	<code>10 < 5</code>	False
>	greater than	<code>10 > 5</code>	True
<=	less than or equal to	<code>126 <= 100</code>	False
>=	greater than or equal to	<code>5.0 >= 5.0</code>	True

Logical Operators

- These operators return true or false

Operator	Example	Result
and	<code>9 != 6 and 2 < 3</code>	True
or	<code>2 == 3 or -1 < 5</code>	True
not	<code>not 7 > 0</code>	False

The `if` Statement

- Syntax:

```
if <condition>:  
    <statements>
```

```
x = 5  
if x > 4:  
    print("x is greater than 4")  
print("Now I'm outside the if")
```


The `if` Statement

- The colon is required for the `if`
- Note that all statements indented one level in from the `if` are within its scope:

```
x = 5
if x > 4:
    print("x is greater than 4")
    print("This is inside the if")
print("All done.")
```

The `if/else` Statement

```
if <condition>:  
    <statements>  
else:  
    <statements>
```

- Note the colon following the `else`
- This works exactly the way you would expect

The for Loop

- Syntax:

```
for variableName in groupOfValues:  
    <statements>
```

- *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.

- Example:

```
for x in range(1, 6):  
    print x, "squared is", x * x
```

Range

- The range function specifies a range of integers:

`range(start, stop)` - the integers between start (**inclusive**) and stop (**exclusive**)

- It can also accept a third value specifying the change between values.

`range(start, stop, step)` - the integers between start (**inclusive**) and stop (**exclusive**) by step

The `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>:  
    <statements>
```

- Example:

```
number = 1  
while number < 200:  
    print (number)  
    number = number * 2
```

Exercise

- Write a Python program to compute and display the first 16 powers of 2, starting with 1 (*hint: $2^{**}1 = 2$*)
- Do this in the Python shell

Exercise

- Write a Python program to compute and display the first 16 powers of 2, starting with 1 (*hint: 2**1 = 2*)
- Do this in the Python shell

```
for x in range(1,17):  
    print (2**x)
```

Strings

- String: A sequence of text characters in a program.
- Strings start and end with quotation mark " or apostrophe ' characters.
- Examples:
"hello"
"This is a string"
"This, too, is a string. It can be very long!"
- A string (enclosed by ") may not span across multiple lines or contain a " character inside it.

"This is not
a legal String."

"This is not a "legal" String either."

'But this is a "legal" String!'

Letting user enter a string into a program

- A user often must be able to type information into a running program
- Use the `input()` statement. You may enter a string inside the parentheses to give the user directions

```
x = input("Enter your name:")  
print(x)  
y = input()  
print(y)
```

IMPORTANT: No matter what the user enters, Python stores it as a **string!**

Strings

- A string can represent characters by preceding them with a backslash.
 - `\t` tab character
 - `\n` new line character
 - `\"` quotation mark character
 - `\\` backslash character
- Example: `"Hello\tthere\nHow are you?"`

Indexing Strings

- You can use square brackets to index a string:

```
name = "Arpita Nigam"
```

```
print(name, " starts with ", name[0])
```

String Functions

- `len(string)` - number of characters in a string
- `str.lower(string)` - lowercase version of a string
- `str.upper(string)` - uppercase version of a string
- `str.isalpha(string)` - True if the string has only alpha chars
- Many others: `split`, `replace`, `find`, `format`, etc.

- Note the “dot” notation: These are static methods.

Byte Arrays and Strings

- Strings are Unicode text and not mutable
- Byte arrays are mutable and contain raw bytes
- For example, reading Internet data from a URL gets bytes
- Convert to string:

```
cmd = response.read()
```

```
strCmd = str(cmd)
```

Other Built-in Types

- tuples, lists, sets, and dictionaries
- They all allow you to group more than one item of data together under one name
- You can also search them

Tuples

- Unchanging Sequences of Data

- Enclosed in parentheses:

```
tuple1 = ("This", "is", "a", "tuple")
```

```
print(tuple1)
```

- This prints the tuple exactly as shown

```
Print(tuple1[1])
```

- Prints "is" (without the quotes)

Lists

- Changeable sequences of data
- Lists are created by using square brackets:

```
breakfast = [ "coffee", "tea", "toast", "egg" ]
```

- You can add to a list:

```
breakfast.append("waffles")
```

```
breakfast.extend(["cereal", "juice"])
```


Dictionaries

- Groupings of Data Indexed by Name
- Dictionaries are created using braces

```
sales = {}
```

```
sales["January"] = 10000
```

```
sales["February"] = 17000
```

```
sales["March"] = 16500
```

- The keys method of a dictionary gets you all of the keys as a list

Sets

- Sets are similar to dictionaries in Python, except that they consist of only keys with no associated values.
- Essentially, they are a collection of data with no duplicates.
- They are very useful when it comes to removing duplicate data from data collections.

Writing Programs

- Bring up the Idle3 IDE
- The first line of your code should be this: `#!/usr/bin/env python 3.1`
- Write the same code you wrote for the exercise of powers of 2 using the IDE's editor
- Press the F5 key to run the program
- It will ask you to save the program. Give it a name like `PowersOf2.py`
- The program will run in a Python shell from the IDE
- If there are errors, the shell will tell you

Writing Functions

- Define a function:

```
def <function name>( <parameter list> )
```

- The function body is indented one level:

```
def computeSquare(x)
```

```
    return x * x
```

```
# Anything at this level is not part of the function
```

Error Handling

- Use try/except blocks, similar to try/catch:

```
fridge_contents = {"egg":8, "mushroom":20,  
"pepper":3, "cheese":2,  
"tomato":4, "milk":13}  
try:  
    if fridge_contents["orange juice"] > 3:  
        print("Sure, let's have some juice!")  
except KeyError:  
    print("Awww, there is no orange juice.")
```

Error Handling

- Note that you must specify the type of error
- Looking for a key in a dictionary that doesn't exist is an error
- Another useful error to know about:

try:

```
    sock = BluetoothSocket(RFCOMM)
```

```
    sock.connect((bd_addr, port))
```

```
except BluetoothError as bt
```

```
    Print("Cannot connect to host: " + str(bt))
```

Using the GPIO Pins

- The Raspberry Pi has a 40-pin header, many of which are general-purpose I/O pins
- Include the library:

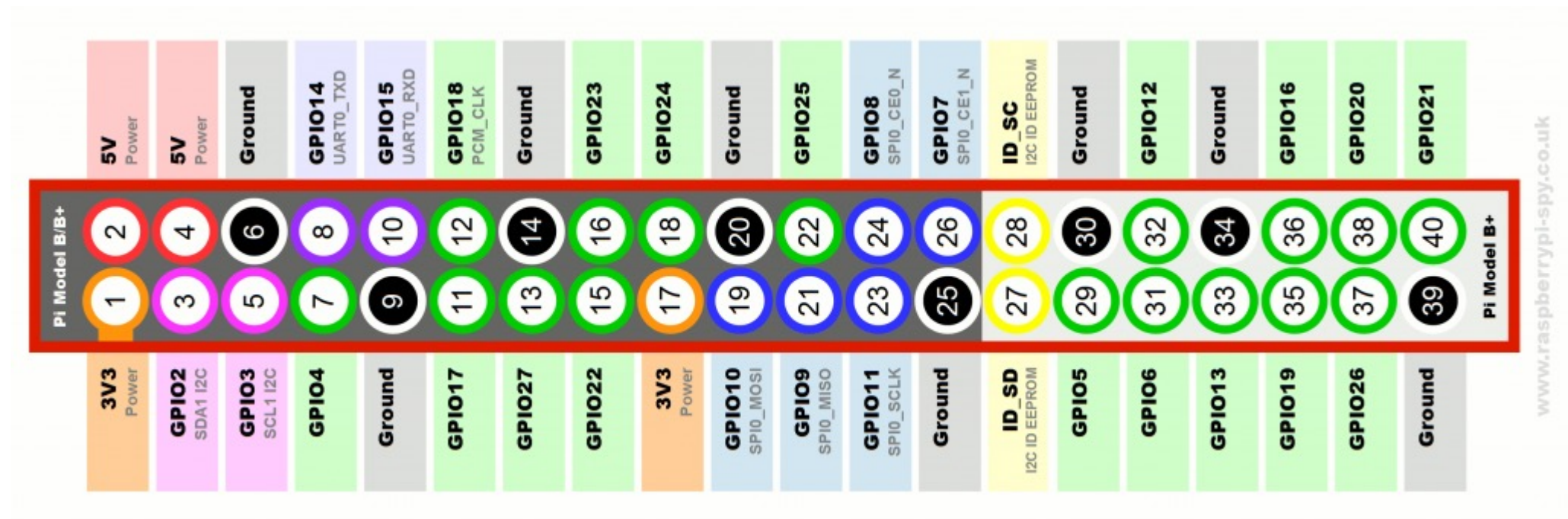
```
import RPi.GPIO as GPIO
```

- Set up to use the pins:

```
GPIO.setmode(GPIO.BOARD)
```

Using the GPIO Pins

- The **GPIO.BCM** option means that you are referring to the pins by the "Broadcom SOC channel" number, these are the numbers after "GPIO" in the green rectangles around the outside of the diagram:



Using the GPIO Pins

- This is from my home-control code:

```
LIGHT = 22
```

```
MOTION = 23
```

```
GPIO.setup(LIGHT, GPIO.OUT)    # For turning on the light
```

```
GPIO.setup(MOTION, GPIO.IN)    # For reading the motion sensor
```

Using the GPIO Pins

- Reading from a GPIO pin:

```
# If we detect motion, print that.  
if GPIO.input(MOTION):  
    print( "Motion detected")
```

Using the GPIO Pins

- Output to GPIO:

```
if cmd=='LIGHTON':  
    GPIO.output(LIGHT, True)    # turn on the light
```

Exercise

- Write a Python program that blinks an LED at a rate of 1 second on, one second off

Python File I/O

- You can read and write text files in Python much as you can in other languages, and with a similar syntax.
- To open a file for reading:

try:

```
    configFile = open(configName, "r")
```

except IOError as err:

```
    print("could not open file: " + str(err))
```

Python File I/O

- To read from a file:

```
while 1:  
    line = configFile.readline()  
    if len(line) == 0:  
        break
```

Python File I/O

- You can also read all lines from a file into a list, then iterate over the list:

```
lines = file.readlines()
for line in lines:
    print(line)
file.close()
```

Python File I/O

- Writing to a text file

```
file=open('test.txt','w')
```

```
file.write("This is how you create a new text file")
```

```
file.close()
```