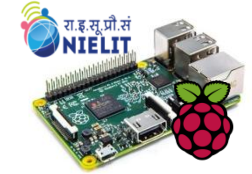


Python

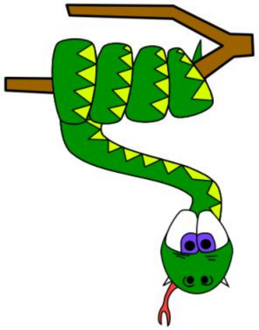
Dr. Sarwan Singh

Deputy Director(S)

NIELIT Chandigarh

Education is the kindling of a flame, not the filling of a vessel.

- Socrates



Programming with Python

Python…

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

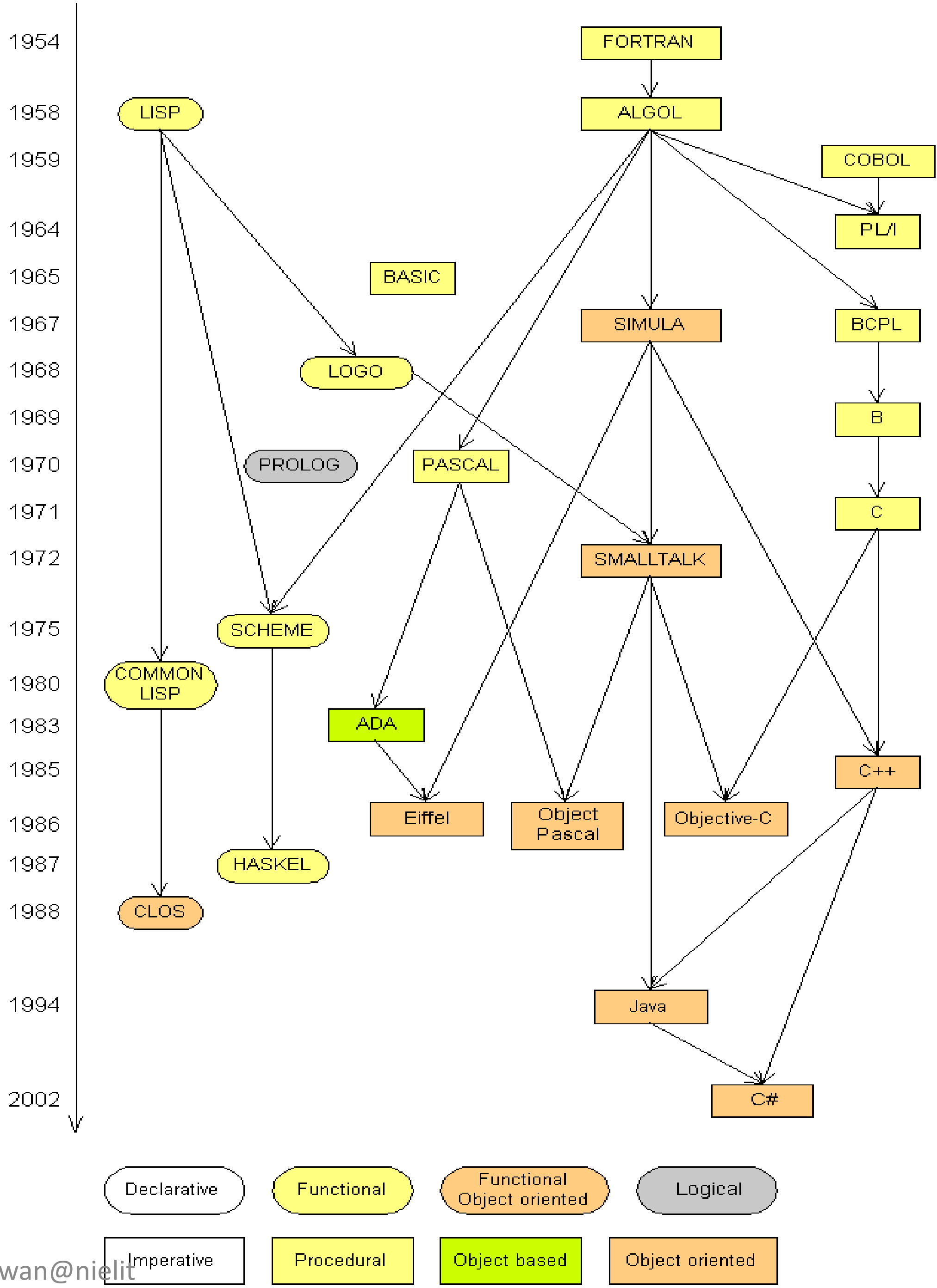
- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers

History of Python

- Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.
- Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).
- Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

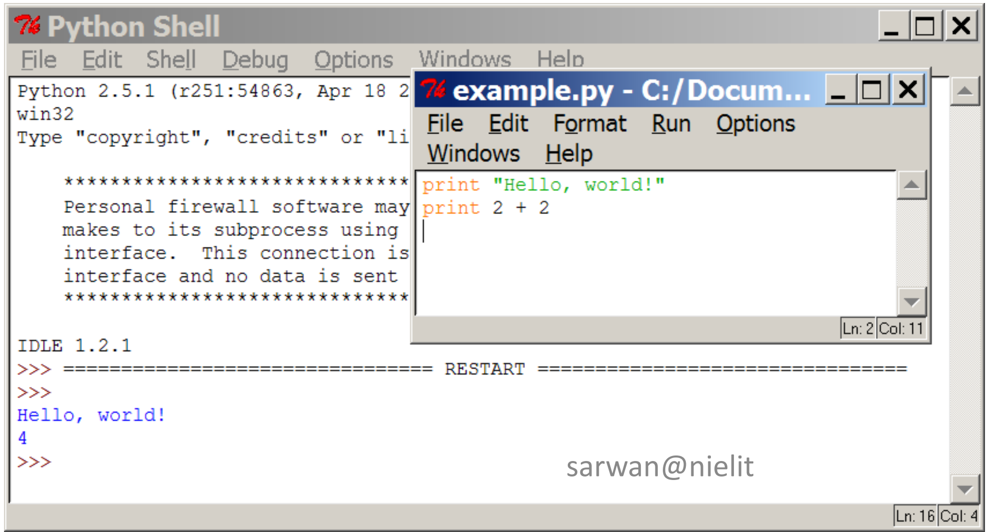
Languages

- Some influential ones
  - FORTRAN
    - science / engineering
  - COBOL
    - business data
  - LISP
    - logic and AI
  - BASIC
    - a simple language



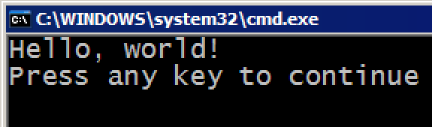
Programming basics

- **code** or **source code**: The sequence of instructions in a program.
- **syntax**: The set of legal structures and commands that can be used in a particular programming language
- **output**: The messages printed to the user by a program
- **console**: The text box onto which output is printed.
  - Some source code editors pop up the console as an external window, and others contain their own console window.



```
Python 2.5.1 (r251:54863, Apr 18 2
win32
Type "copyright", "credits" or "li

    ******************************
    Personal firewall software may
    makes to its subprocess using
    interface.  This connection is
    interface and no data is sent
    ******************************

IDLE 1.2.1
>>> ================================ RESTART ================================
>>>
Hello, world!
4
>>>
```

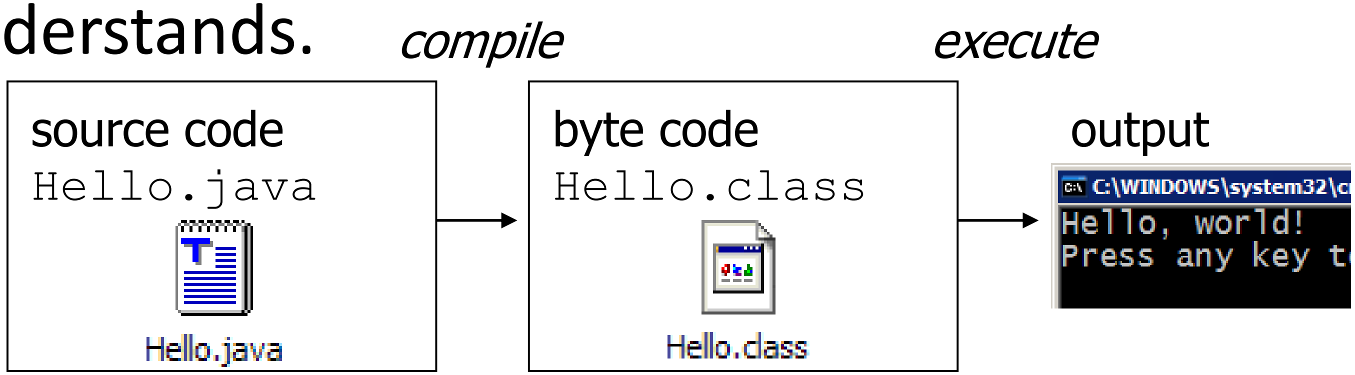
```
print "Hello, world!"
print 2 + 2
```



```
Hello, world!
Press any key to continue
```

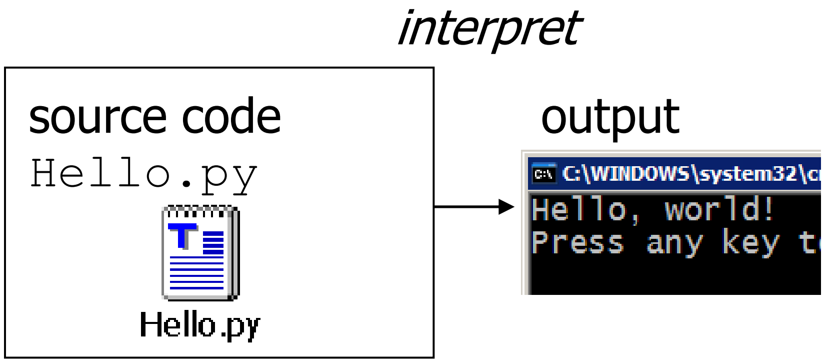
Compiling and interpreting

- Many languages require you to *compile* (translate) your program into a form that the machine understands.



```
Hello, world!
Press any key t
```

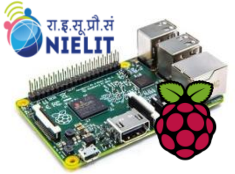
- Python is instead directly *interpreted* into machine instructions.



```
Hello, world!
Press any key t
```

What Is Python?

- Created in 1990 by Guido van Rossum
  - While at CWI, Amsterdam
  - Now hosted by centre for national research initiatives, Reston, VA, USA
- Free, open source
  - And with an amazing community
- Object oriented language
  - “Everything is an object”
- No separate compilation step - compiled version is cached when used.



Why Python?

- Designed to be easy to learn and master
  - Clean, clear syntax
  - Very few keywords
- Highly portable
  - Runs almost anywhere - high end servers and workstations, down to windows CE
  - Uses machine independent byte-codes
- Extensible
  - Designed to be extensible using C/C++, allowing access to many external libraries

Python: a modern hybrid

- A language for scripting and prototyping
- Balance between extensibility and powerful built-in data structures
- genealogy:
  - Setl (NYU, J.Schwartz et al. 1969-1980)
  - ABC (Amsterdam, Meertens et al. 1980-)
  - Python (Van Rossum et all. 1996-)
- Very active open-source community

Prototyping

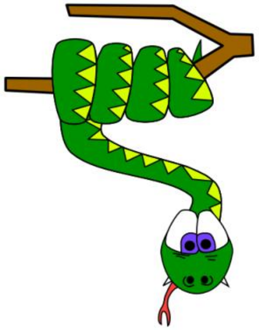
- Emphasis on experimental programming:
- Interactive (like LISP, ML, etc).
- Translation to bytecode (like Java)
- Dynamic typing (like LISP, SETL, APL)
- Higher-order function (LISP, ML)
- Garbage-collected, no ptrs
  (LISP, SNOBOL4)

Prototyping

- Emphasis on experimental programming:
- Uniform treatment of indexable structures (like SETL)
- Built-in **associative structures** (like SETL, SNOBOL4, Postscript)
- Light syntax, indentation is significant (from ABC)

Most obvious and notorious features

- Clean syntax plus high-level data types
  - Leads to fast coding
- Uses white-space to delimit blocks
  - Humans generally do, so why not the language?
  - Try it, you will end up liking it
- Variables do not need declaration
  - Although not a type-less language



Data type, expression

Operators

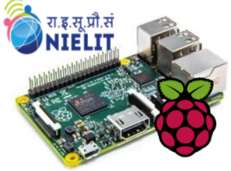
- Python has many operators. Some examples are:

`+, -, *, /, %, >, <, ==`
`print`

- Operators perform an action on one or more operands. Some operators accept operands before and after themselves:

`operand1 + operand2`, or `3 + 5`

- Others are followed by one or more operands until the end of the line, such as: `print “Hi!”, 32, 48`
- When operators are evaluated, they perform action on their operands, and produce a new value.



Example Expression Evaluations

- An expression is any set of values and operators that will produce a new value when evaluated. Here are some examples, along with the new value they produce when evaluated:

`5 + 10`	produces	15
`“Hi” + “ “ + “Jay!”`	produces	“Hi Jay!”
`10 / (2+3)`	produces	2
`10 > 5`	produces	True
`10 < 5`	produces	False
`10 / 3.5`	produces	2.8571428571
`10 // 3`	produces	3
`10 % 3`	produces	1

List of Operators

+, -, *, /, <, >, <=, >=, ==, %, //

- Some operators should be familiar from the world of mathematics such as Addition (+), Subtraction (-), Multiplication (*), and Division (/).
- Python also has comparison operators, such as Less-Than (<), Greater-Than (>), Less-Than-or-Equal(<=), Greater-Than-or-Equal (>=), and Equality-Test (==). These operators produce a True or False value.
- A less common operator is the Modulo operator (%), which gives the remainder of an integer division. 10 divided by 3 is 9 with a remainder of 1:

`10 // 3` produces 3, while `10 % 3` produces 1

DANGER! Operator Overloading!

- NOTE! Some operators will work in a different way depending upon what their operands are. For example, when you add two numbers you get the expected result: `3 + 3` produces 6.
- But if you “add” two or more strings, the + operator produces a concatenated version of the strings: `“Hi” + “Jay”` produces “HiJay”
- Multiplying strings by a number repeats the string!

`“Hi Jay” * 3` produces “Hi JayHi JayHiJay”

- The % sign also works differently with strings:

`“test %f” % 34` produces “test 34”

Data Types

- In Python, all data has an associated data “Type”.
- You can find the “Type” of any piece of data by using the type() function:

type(“Hi!”) produces <type 'str'>

type(True) produces <type 'bool'>

type(5) produces <type 'int'>

type(5.0) produces <type 'float'>

- Note that python supports two different types of numbers, Integers (int) and Floating point numbers (float). Floating Point numbers have a fractional part (digits after the decimal place), while Integers do not!

Effect of Data Types on Operator result

- Math operators work differently on Floats and Ints:
  - int + int produces an int
  - int + float or float + int produces a float
- This is especially important for division, as integer division produces a different result from floating point division:

`10 // 3` produces 3

`10 / 3` produces 3.3333

`10.0 / 3.0` produces 3.3333333

- Other operators work differently on different data types: + (addition) will add two numbers, but concatenate strings.

Data types in Python

- The simple data types in Python are:
- Numbers
  - int – Integer: -5, 10, 77
  - float – Floating Point numbers: 3.1457, 0.34
- bool – Booleans (True or False)
- Strings are a more complicated data type (called Sequences) that we will discuss more later. They are made up of individual letters (strings of length 1)

Type Conversion

- Data can sometimes be converted from one type to another. For example, the string “3.0” is equivalent to the floating point number 3.0, which is equivalent to the integer number 3
- Functions exist which will take data in one type and return data in another type.
    - `int()` - Converts compatible data into an integer. This function will truncate floating point numbers
    - `float()` - Converts compatible data into a float.
    - `str()` - Converts compatible data into a string.
- Examples:

`int(3.3)` produces 3

`float(3)` produces 3.0

`int(“7”)` produces 7

`int(“7.1”)` throws an ERROR!

`float(“Test”)` Throws an ERROR!

`str(3.3)` produces “3.3”

`float(“3.5”)` produces 3.5

Variables

- Variables are names that can point to data.
- They are useful for saving intermediate results and keeping data organized.
- The assignment operator (=) assigns data to variables.
  - Don't confuse the assignment operator (single equal sign, =) with the Equality-Test operator (double equal sign, ==)
- Variable names can be made up of letters, numbers and underscores (_), and must start with a letter.

Variables

- When a variable is evaluated, it produces the value of the data that it points to.
- For example:

  `myVariable = 5`

  `myVariable` produces 5

  `myVariable + 10` produces 15
- Value MUST be assigned to a variable (to create the variable name) before using (evaluating) it.

Math commands

- Python has useful commands for performing calculations.

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

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

- To use many of these commands, you must write the following at the top of your Python program:

```
from math import *
```

print

- `print` : Produces text output on the console.
- Syntax: `print` “***Message***” ***or*** ***Expression***
  - Prints the given text message or expression value on the console, and moves the cursor down to the next line.

  `print` ***Item1*** **,** ***Item2*** **,** **...** **,** ***ItemN***

  - Prints several messages and/or expressions on the same line.
- Examples:

```
print "Hello, world!"
age = 45
print "You have", 65 - age, "years to retire"
```

Output:

```
Hello, world!
You have 20 years to retire
```

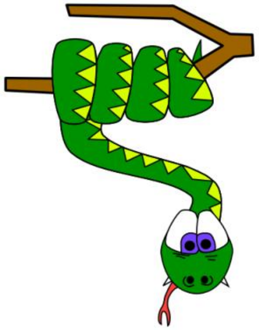
input

- `input` : Reads a number from user input.
  - You can assign (store) the result of `input` into a variable.
  - Example:

```
age = input("How old are you? ")
print "Your age is", age
print "You have", 65 - age, "years to
   retire"
```

Output:

```
How old are you? 53
Your age is 53
You have 12 years retire
```



Function

Program Example

- Find the area of a circle given the radius:

```
Radius = 10
pi = 3.14159
area = pi * Radius * Radius
print( area )
```

will print 314.15 to the screen.

Code Abstraction & Reuse Functions

If something is to be done (like calculate the area of a circle) multiple times, code can be encapsulated inside a *Function*.

“A Function is a named sequence of statements that perform some useful operation.”

Functions may or may not take parameters, and may or may not return results.

Syntax:

```
def NAME( LIST OF PARAMETERS):
    STATEMENTS
    STATEMENTS
```

How to use a function

- function calling :

```
functionName( Parameters)
```

- function returning values :

```
returnResult =
  functionName(Parameters)
```

Indentation is IMPORTANT!

- A function is made up of two main parts, the Header, and the Body.
- The function header consists of:

  ```
  def funcName(param1,param2):
  ```

  - def keyword
  - function name
  - zero or more parameters, comma separated, inside of parenthesis ()
  - A colon :
- The function body consists of all statements in the block that directly follows the header.
- A block is made up of statements that are at the same indentation level.

Function- naive example

```
def findArea( ):
    Radius = 10
    pi = 3.1459
    area = pi * Radius * Radius
    print(area)
```

- This function will ONLY calculate the area of a circle with a radius of 10!
- This function will PRINT the area to the screen, but will NOT return the value pointed to by the area variable.

Function with syntax error!

```
def findArea( ):
   Radius = 10
  pi = 3.1459
 area = pi * Radius * Radius
   print(area)
```

- DONOT mix indentation levels within the same block! The above code will result in a **syntax error!**

Function with arguments

```
def findArea( Radius ):
   pi = 3.1459
   area = pi * Radius * Radius
   print( area)
```

- This function will work with any sized circle!
- This function will PRINT the area to the screen, but will NOT return the value pointed to by the area variable.

Function returning value

```
def findArea( Radius ):
    pi = 3.1459
    area = pi * Radius * Radius
    return area
```

- This function will work with any sized circle!
- This function will return the area found, but will NOT print it to the screen. If we want to print the value, we must print it ourselves:

  ```
  circleArea = findArea(15)
  print circleArea
  ```

- Note the use of the circleArea variable to hold the result of our findArea function call.

Keywords, Name-spaces & Scope

- In Python, not all names are equal.
- Some names are reserved by the system and are already defined. Examples are things like: def, print, if, else, while, for, in, and, or, not, return. These names are built in keywords.
- Names that are defined in a function are “local” to that function.
- Names that are defined outside of a function are “global” to the module.
- Local names overshadow global names when inside the function that defined them.
- If you want to access a global variable from inside of a function, you should declare it “global”.

Global vs Local example

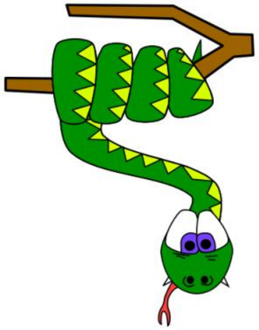
```
myVariable = 7
myParam = 20

def func1(myParam):
    myVariable = 20
    print(myParam)

func1(5)
print(myVariable)
```

- What gets printed? 5 and 7
- The “local” myVariable inside func1 is separate from (and overshadows) the “global” myVariable outside of func1
- The “local” myParam inside func1 is different from the “global” myParam defined at the top.

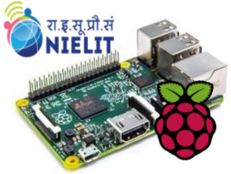
Global vs Local example – part 2

```
myVariable = 7
myParam = 20

def func1(myParam):
    global myVariable
    myVariable = 20
    print(myParam)

func1(5)
print(myVariable)
```

- What gets printed? 5 and 20
- The “local” myVariable inside func1 is separate from the “global” myVariable outside of func1
- The function assigns 20 to the “global” myVariable, overwriting the 7 before it gets printed.



Repetition (loops)
and Selection (if/else)



Controlling Program Flow

- To make interesting programs, you must be able to make decisions about data and take different actions based upon those decisions.
- The IF statement allows you to conditionally execute a block of code.
- The syntax of the IF statement is as follows:

```
 if  boolean_expression :
     STATEMENT
     STATEMENT
```

- The indented block of code following an if statement is executed if the boolean expression is true, otherwise it is skipped.

IF statement - example

```
numberOfWheels = 3
if ( numberOfWheels < 4):
    print(“You don't have enough
  wheels!”)
    print(“I'm giving you 4 wheels!”)
    numberOfWheels = 4

print(“You now have”, numberOfWheels,
  “wheels”)
```

- The last print statement is executed no matter what. The first two print statements and the assignment of 4 to the numberOfWheels is only executed if numberOfWheels is less than 4.

IF/ELSE

- If you have two mutually exclusive choices, and want to guarantee that only one of them is executed, you can use an IF/ELSE statement. The ELSE statement adds a second block of code that is executed if the boolean expression is false.

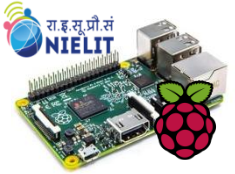
```
if  boolean_expression :
    STATEMENT
    STATEMENT
else:
    STATEMENT
    STATEMENT
```

IF/ELSE statement - example

```
numberOfWheels = 3
if ( numberOfWheels < 3):
    print(“You are a motorcycle!”)
else:
    print(“You are a Car!”)

print(“You have”, numberOfWheels,
 “wheels”)
```

- The last print statement is executed no matter what. If numberOfWheels is less than 3, it's called a motorcycle, otherwise it's called a car!

IF/ELIF/ELSE



- If you have several mutually exclusive choices, and want to guarantee that only one of them is executed, you can use an IF/ELIF/ELSE statements. The ELIF statement adds another boolean expression test and another block of code that is executed if the boolean expression is true.

```
if  boolean_expression :
   STATEMENT
   STATEMENT
elif 2nd_boolean_expression ):
   STATEMENT
   STATEMENT
else:
   STATEMENT
   STATEMENT
```

IF/ELSE statement - example

```
numberOfWheels = 3
if ( numberOfWheels == 1):
    print(“You are a Unicycle!”)
elif (numberOfWheels == 2):
    print(“You are a Motorcycle!”)
elif (numberOfWheels == 3):
    print(“You are a Tricycle!”)
elif (numberOfWheels == 4):
    print(“You are a Car!”)
else:
    print(“That's a LOT of
 wheels!”)
```

- Only the print statement from the first true boolean expression is executed.

Getting input from the User

- Your program will be more interesting if we obtain some input from the user.
- But be careful! The user may not always give you the input that you wanted, or expected!
- A function that is useful for getting input from the user is:

`input(<prompt string>)` - always returns a string

- You must convert the string to a float/int if you want to do math with it!

Input Example – possible errors from the input() function

- `userName = input(“What is your name?”)`
- `userAge = int( input(“How old are you?”) )`
- `birthYear = 2007 - userAge`

- `print(“Nice to meet you, “ + userName)`
- `print(“You were born in: “, birthYear)`

- `input()` is guaranteed to give us a string, no matter WHAT the user enters.
- But what happens if the user enters “ten” for their age instead of 10?

Input Example – possible errors from the input() function

```
• userName = raw_input(“What is your
  name?”)
• userAge = input(“How old are you?”)
• try:
•    userAgeInt = int(userAge)
• except:
•    userAgeInt = 0
• birthYear = 2010 - userAgeInt

• print(“Nice to meet you, “ + userName)
• if userAgeInt != 0:
•     print(“You were born in: “,
  birthYear )
```

- The try/except statements protects us if the user enters something other than a number. If the int() function is unable to convert whatever string the user entered, the except clause will set the userIntAge variable to zero.

Repetition can be useful!

- Sometimes you want to do the same thing several times.
- Or do something very similar many times.
- One way to do this is with repetition:

```
print 1
print 2
print 3
print 4
print 5
print 6
print 7
print 8
print 9
print 10
```

Looping, a better form of repetition.

- Repetition is OK for small numbers, but when you have to do something many, many times, it takes a very long time to type all those commands.
- We can use a loop to make the computer do the work for us.
- One type of loop is the “while” loop. The while loop repeats a block of code until a boolean expression is no longer true.

Syntax:

```
while boolean expression :
     STATEMENT
     STATEMENT
     STATEMENT
```

How to STOP looping!

- It is very easy to loop forever:
- while True :
-    print(“again, and again, and again”)

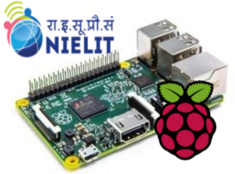
- The hard part is to stop the loop!
- Two ways to do that is by using a loop counter, or a termination test.
  - A loop counter is a variable that keeps track of how many times you have gone through the loop, and the boolean expression is designed to stop the loop when a specific number of times have gone bye.
  - A termination test checks for a specific condition, and when it happens, ends the loop. (But does not guarantee that the loop will end.)

Loop Counter

```
timesThroughLoop = 0

while (timesThroughLoop < 10):
   print(“This is time”,
      timesThroughLoop, in the loop.”)
   timesThroughLoop = timesThroughLoop+1
```

- Notice that we:
  - Initialize the loop counter (to zero)
  - Test the loop counter in the boolean expression (is it smaller than 10, if yes, keep looping)
  - Increment the loop counter (add one to it) every time we go through the loop
- If we miss any of the three, the loop will NEVER stop!

While loop example, with a termination test



- Keeps asking the user for their name, until the user types “quit”.

```
keepGoing = True
while ( keepGoing):
   userName = input(“Enter your name! (or
                      quit to exit)” )
   if userName == “quit”:
       keepGoing = False
   else:
      print(“Nice to meet you,“ + userName)
print(“Goodbye!”)
```

The `for` loop

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

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

  - We indent the statements to be repeated with tabs or spaces.
  - ***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 `range` function.
  - Example:

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

Output:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

`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***

range

- Example:

```
for x in range(5, 0, -1):
    print x
print "Blastoff!"
```

Output:

```
5
4
3
2
1
Blastoff!
```

-
```
>>> for i in range(3):
...     print i
...
0
1
2
```

Exceptions

- Python uses exceptions for errors
  - `try` / `except` block can handle exceptions
-
```
>>> try:
...     1/0
... except ZeroDivisionError:
...     print "Eeek"
...
Eeek
>>>
```

Exceptions

- **`try`** / **`finally`** block can guarantee execute of code even in the face of exceptions
-
```
>>> try:
...     1/0
... finally:
...     print "Doing this anyway"
...
Doing this anyway
Traceback (innermost last):  File "<interactive
input>", line 2, in ?
ZeroDivisionError: integer division or modulo
>>>
```

Threads

- Number of ways to implement threads
- Highest level interface modelled after Java
-

```
>>> class DemoThread(threading.Thread):
...     def run(self):
...         for i in range(3):
...             time.sleep(3)
...             print i
...
>>> t = DemoThread()
>>> t.start()
>>> t.join()
0
1 <etc>
```