MALAYAN COLLEGES LAGUNA
COLLEGE OF COMPUTER AND INFORMATION SCIENCE

Week 1: Computational Problem Solving

2019 – 2020 1st Term

Ms. Rhea N. Tortor - IT

Topics:

- Process of Computational Problem Solving
  - Problem Analysis, Design, Implementation, and Testing
  - Algorithms
  - Pseudocodes and Flowcharting

Learning Objectives

- Identify the terminologies in algorithm, pseudocode, and flowcharts. (CO1)
- Use appropriate flowcharting symbols in solving specific problem. (CO1)

TERMINOLOGIES

- **Algorithm**
  - the step-by-step sequence of instructions that describe how the data is to be processed to produce the desired output
- **Logic**
  - Art of reasoning

TERMINOLOGIES

- **Syntax**
  - Rules governing usage of words and punctuation
- **Semantics**
  - Rules governing the logic or idea of languages
- **Machine Language**
  - The computer's native language represented as a series of 0's and 1's (binary form)

TERMINOLOGIES

- Compiler or Interpreter
  - Translation software that converts programmer's statements to binary form notifies the programmer if the programming language is used incorrectly
- Compiler
  - Entire program is translated before it can execute
- Interpreter
  - Each instruction is translated just prior to execution

PROGRAMMING AS PROBLEM SOLVING

PROGRAMMING AS PROBLEM SOLVING

- One definition of programming is it is applied problem solving
  - You have a problem (e.g. need a program to calculate the area of the circle).
  - What inputs and outputs are needed?
  - How does what is entered produce the right output?
  - What needs to be done?

ANALYZE THE PROBLEM

- Thoroughly understand the problem
- Understand problem requirements
  - Does program require user interaction?
  - Does program manipulate data?
  - What is the output?
- If the problem is complex, divide it into sub problems
  - Analyze each sub problem as above

BASIC PROPERTIES OF THE ALGORITHMS

- **Generality**
  - An algorithm should be designed for an entire class of problems not only for a particular instance of a problem.
- **Finiteness**
  - An algorithm should be described by a finite structure and each of the operations that it contains have to executed in a finite amount of time.

BASIC PROPERTIES OF THE ALGORITHMS

- **Non-ambiguity**
  - The operations in an algorithm have to be rigorously specified, i.e. without ambiguities. At every step of an algorithm one have to know exactly which is the next operation which will be executed.
- **Efficiency**
  - The algorithms are useful only if they need a reasonable amount of computing resources. By computing resources we mean both memory and time resources

REPRESENTATION OF ALGORITHMS

- A single algorithm can be represented in many ways:
  - Formulas: F = (9/5)C + 32
  - Words: Multiply the Celsius by 9/5 and add 32.
  - Flow Charts.
  - Pseudo-code.
- In each case, the algorithm stays the same; the implementation differs!

REPRESENTATION OF ALGORITHMS

- A program is a representation of an algorithm designed for computer applications.
- Process: Activity of executing a program, or execute the algorithm represented by the program
  - Process: Activity of executing an algorithm.

PROGRAMMING

PROCESS

PROGRAMMING PROCESS

1. Understanding the problem
2. Planning the logic
3. Coding the program
   - Using software to translate the program into machine language
4. Testing the program
5. Putting the program into production

1. UNDERSTANDING THE TASK

- What problem do you have to solve anyway?
- What process are you trying to streamline?
- What error are you trying to fix?

Example:

www.nasa.gov, ***at the National Aeronautics and Space Administration, scientists have to understand every detail in a problem. Any mistake will certainly result in a disaster. Launching a rocket into space is certainly something complicated. All the scientists have to cooperate in making a successful rocket launch***

Failure to analyze and understand the requirements leads to a failed solution!

2. PLANNING THE LOGIC

- The heart of the programming process lies in planning the program's logic.
- Once you understand your problem, you have to devise a plan to solve it. Part of your plan includes the selection of an appropriate programming language.

- **Develop a Solution**
  - Algorithm: the exact steps used to solve a problem
- **Two most common planning tools:**
  - Flowcharts
  - Pseudocode

3. CODING THE PROGRAM

- Coding is one of the most tedious parts in programming.
- Programmers choose a particular language because some languages have built-in capabilities to make them more efficient than others at handling certain types of operations.

- **Code the Solution**
  - Also called writing the program, and
  - implementing the solution
  - Program should contain well-defined patterns or structures of the following types:
    - Sequence
    - Selection
    - Iteration

4. TESTING AND RUNNING THE PROGRAM

- Once a program has been coded, compiled and – or interpreted, you can test if the program will run or not.
- Some compilers and interpreters will enable you to debug a code when an error is apparent.

- **Test and Correct the Program**
  - **Testing**: method to verify correctness and that requirements are met
  - **Bug**: a program error
  - **Debugging**: the process of locating an error, and correcting and verifying the correction
  - **Testing** may reveal errors, but does not guarantee the absence of errors

5. PUTTING THE PROGRAM INTO PRODUCTION

- **Backup:** process of making copies of program code and documentation on a regular basis
- Backup copies = insurance against loss or damage
  - Consider using off-site storage for additional protection

PROBLEM SOLVING STRATEGIES

PROBLEM SOLVING STRATEGIES

- Working backwards
  - Reverse engineering is taking apart an object to see how it works in order to duplicate or enhance the object. Once you know it can be done, it is much easier to do
- Look for a related problem that has been solved before
- Stepwise Refinement
  - Break the problem into several sub-problems
  - Solve each sub problem separately
  - Produces a modular structure

STEPWISE REFINEMENT

- Stepwise refinement is a top-down methodology in that it progresses from the general to the specific.
  - Bottom-up methodologies progress from the specific to the general.
  - Solutions produced by stepwise refinement posses a natural modular structure - hence its popularity in algorithmic design.

OBJECT-ORIENTED DESIGN METHODOLOGY

- Four stages to the decomposition process
  - Brainstorming
  - Filtering
  - Scenarios
  - Responsibility algorithms

STEP 1: BRAINSTORMING

- A group problem-solving technique that involves the spontaneous contribution of ideas from all members of the group
  - All ideas are potential good ideas
  - Think fast and furiously first, and ponder later
  - A little humor can be a powerful force
- Brainstorming is designed to produce a list of candidate classes

STEP 2: FILTERING

- Determine which are the core classes in the problem solution
- There may be two classes in the list that have many common attributes and behaviors
- There may be classes that really don’t belong
- in the problem solution

STEP 3: SCENARIOS

- Assign responsibilities to each class
- There are two types of responsibilities
  - What a class must know about itself (knowledge)
  - What a class must be able to do (behavior)
- Encapsulation is the bundling of data and actions in such a way that the logical properties of the data and actions are separated from the implementation details

STEP 4: RESPONSIBILITY ALGORITHMS

- The algorithms must be written for the responsibilities
  - Knowledge responsibilities usually just return the contents of one of an object's variables
  - Action responsibilities are a little more complicated, often involving calculations

PSEUDOCODE STATEMENTS AND FLOWCHART SYMBOLS

PSEUDOCODE

- An English-like representation of the same thing.

PSEUDO = FALSE

CODE = A program means to put in a programming language

PSEUDOCODE = "FALSE CODE"

RULES FOR PSEUDOCODE

- The pseudocode must be language independent. Try to avoid the use of words peculiar to any programming language.
- Show key words in CAPITAL LETTERS.

```
Example:    IF condition THEN
                DO action
                ENDDO
            ENDIF
```

- Indent lines to make the pseudocode easy to read and understand.

RULES FOR PSEUDOCODE

- Punctuation is optional.
- Every IF must end with an ENDIF.
- Every DO, DO FOR and DO WHILE must end with ENDDO.
- The main routine (the one that goes from START to STOP) has to be shown first. All other routines are to follow.

ADVANTAGES IN USING PSEUDOCODE

- It bridges the gap between human language and computer language.
- It is an intermediate notation that allows expression of program logic in a straight forward, easy to understand manner without concerning the programmer with syntax details.
- It is easier to make changes to pseudocode than to a source program in a high-level language.

Flowchart

- A pictorial representation of the logical steps it takes to solve a problem.

- A **flowchart** is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution of a problem.

Guidelines in Drawing Flowcharts

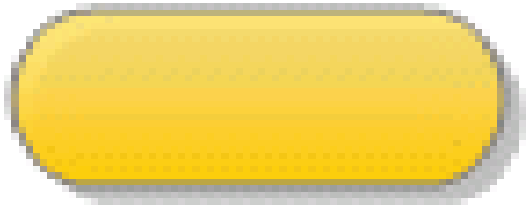
- In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
- The flowchart should be clear, neat and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
- The usual direction of the flow of a procedure or system is from left to right or top to bottom.
- Only one flow line should come out from a process symbol.

- Only one flow line should enter a decision symbol, but two flow lines, one for each possible answer, should leave the decision symbol.
- Only one flow line is used in conjunction with terminal symbol.
- If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. Avoid the intersection of flow lines if you want to make it more effective and better way of communication.
- Ensure that the flowchart has a logical start and finish.
- It is useful to test the validity of the flowchart by passing through it with a simple test data.

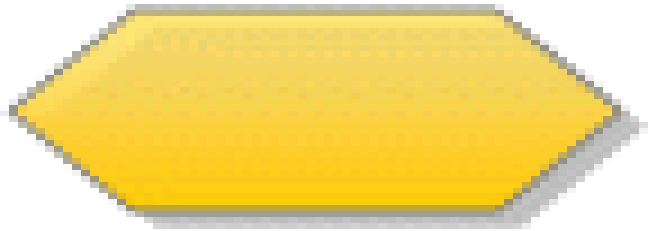
Flowchart Symbols

- Flowcharts are usually drawn using some standard symbols; however, some special symbols can also be developed when required. Some standard symbols, which are frequently, required for flowcharting many computer programs are the following:

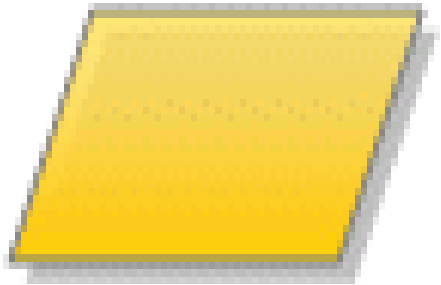
- **Terminal Symbol** - shows the start and stop points in a process.



- **Preparation Symbol** - as the names states, any process step that is a preparation process flow step, such as a set-up operation.



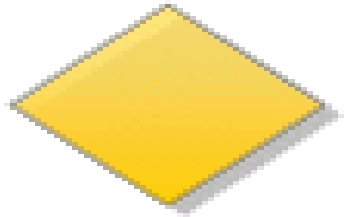
- **Input/output Symbol** - indicates inputs to and outputs from a process.



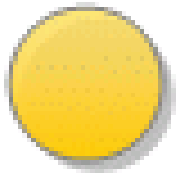
- **Process Symbol** - shows a process or action step. This is the most common symbol in both process flowcharts and business process maps.

- **Predefined Process (Subroutine) Symbol** - is a marker for another process step or series of process flow steps that are formally defined elsewhere. This shape commonly depicts sub-processes (or subroutines in programming flowcharts).

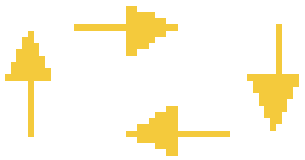
- **Decision Symbol** - Indicates a question or branch in the process flow. Typically, a decision flowchart shape is used when there are 2 options (Yes/No, No/No-Go, etc.)



- **On-Page Connector Symbol** - to show a jump from one point in the process flow to another.

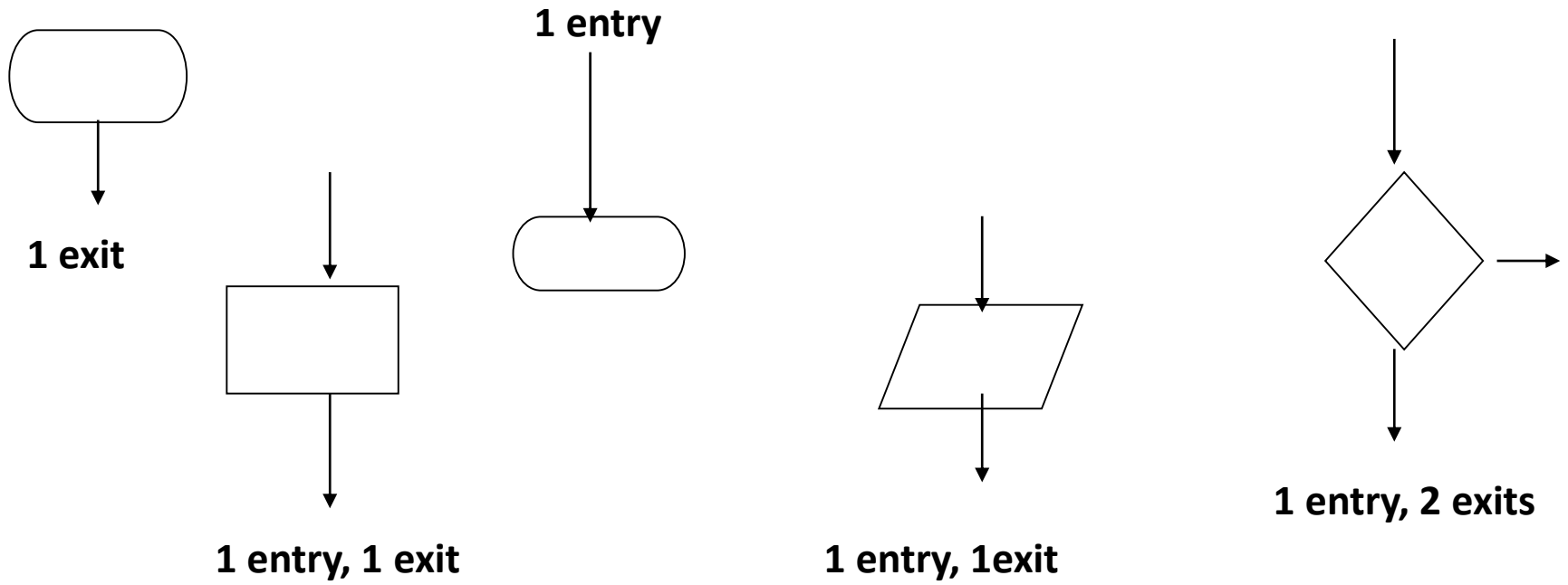


- **Flow lines** – a flow of control arrows indicating the sequence of steps.



Entries and Exits

- The following will show you the number of entries and exits from each symbol.



Summary

- Programming as Problem Solving
- Programming Process
  - Understanding the problem
  - Planning the logic
  - Coding the program
  - Testing the program
  - Putting the program into production
- Problem Solving Strategies
  - Stepwise Refinement

Summary

- Object Oriented Design Methodology
  - Brainstorming
  - Filtering
  - Scenarios
  - Responsibility algorithms
- PSEUDOCODE STATEMENTS
- FLOWCHART Symbols

END

MALAYAN COLLEGES LAGUNA
COLLEGE OF COMPUTER AND INFORMATION SCIENCE

Week 1: Problem Solving Using Python

2019 – 2020 1st

Ms. Rhea N. Tortor - IT

Topics:

- Problem Solving using Python
  - Structure of a Python Program
  - Naming conventions, Python operators, expressions, data types, declarations, assignment, and built-in functions

Learning Objectives

- Explain programming terminologies, data types, input, and output in Python. (CO3)
- Use proper naming convention, appropriate data types, operators, expressions/statements and declarations in Python. (CO3, CO4)

What is Python?

- Python is a popular programming language. It was created in 1991 by Guido van Rossum.
- It is used for:
  - web development (server-side),
  - software development,
  - mathematics,
  - system scripting.

What can Python do?

- can be used on a server to create web applications.
- can be used alongside software to create workflows.
- can connect to database systems. It can also read and modify files.
- can be used to handle big data and perform complex mathematics.
- can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-orientated way or a functional way.

Python Indentations

- very important
- indicate a block of code.
- Example:

```
if 5 > 2:
   print("Five is greater than two!")
```

Python will give you an error if you skip the indentation:

- Example:

```
if 5 > 2:
print("Five is greater than two!")
```

Comments

- Python has commenting capability for the purpose of in-code documentation.
- Comments start with a #, and Python will render the rest of the line as a comment
- Example:

```
#This is a comment.
print("Hello, World!")
```

Docstrings

- Python also has extended documentation capability, called docstrings.
- Docstrings can be one line, or multiline.
- Python uses triple quotes at the beginning and end of the docstring
- Example:

```
"""This is a
multiline docstring."""
print("Hello, World!")
```

Creating Variables

- Unlike other programming languages, Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- Example:

```
x = 5
y = "John"
print(x)
print(y)
```

- Variables do not need to be declared with any particular type and can even change type after they have been set.
- Example:

```
x = 4 # x is of type int
x = "Sally" # x is now of type
str
print(x)
```

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Remember that variables are case-sensitive

Print() statement

- used to output variables
- To combine both text and a variable, Python uses the + character:
- Example:

```
x = "awesome"
print("Python is " + x)
```

- Output:

```
Python is awesome
```

Print() statement (cont.)

- For numbers, the + character works as a mathematical operator:
- Example:

```
x = 5
y = 10
print(x + y)
```

- Output:

```
15
```

If you try to combine a string and a number, Python will give you an error

Example:

```
x = 5
y = "John"
print(x + y)
```

Python Numbers

- There are three numeric types in Python:
  - int
  - float
  - Complex
- To verify the type of any object in Python, use the *type()* function

Int

- Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.
- Example:

```
x = 1
y = 35656222554887711
z = -3255522

print(type(x))
print(type(y))
print(type(z))
```

Output:

```
<class 'int'>
<class 'int'>
<class 'int'>
```

float

- Float, or "floating point number" is a number, positive or negative, containing one or more decimals.
- Example:

```
x = 1.10
y = 1.0
z = -35.59

print(type(x))
print(type(y))
print(type(z))
```

Output:

```
<class 'float'>
<class 'float'>
<class 'float'>
```

float (cont.)

- Float can also be scientific numbers with an "e" to indicate the power of 10.
- Example:

```
x = 35e3
y = 12E4
z = -87.7e100

print(type(x))
print(type(y))
print(type(z))
```

Output:

```
<class 'float'>
<class 'float'>
<class 'float'>
```

Specify a Variable Type

- There may be times when you want to specify a type on to a variable.
- This can be done with casting.
- Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

Type Casting

- Casting in python is therefore done using constructor functions:
  - *int()* - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number) literal, or a string literal (providing the string represents a whole number)
  - *float()* - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
  - *str()* - constructs a string from a wide variety of data types, including strings, integer literals and float literals

int()

- **Example:**

```
x = int(1)    # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

float()

- **Example:**

```
x = float(1)      # x will be 1.0
y = float(2.8)    # y will be 2.8
z = float("3")    # z will be 3.0
w = float("4.2") # w will be 4.2
```

string()

- **Example:**

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

String Literals

- String literals in python are surrounded by either single quotation marks, or double quotation marks.

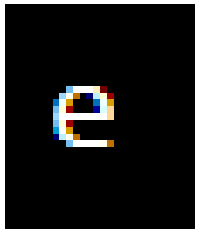
`'hello'` is the same as `"hello"`.

- Strings can be output to screen using the print function.
- For example: `print("hello")`.

- Square brackets can be used to access elements of the string.
- Example:

```
a = "Hello, World!"
print(a[1])
```

Output:



Substring

- Get the characters from position n to position m (not included)
- Example:

```
b = "Hello, World!"
print(b[2:5])
```

Output:

strip()

- removes any whitespace from the beginning or the end
- Example:

Output:

```
Hello, World!
```

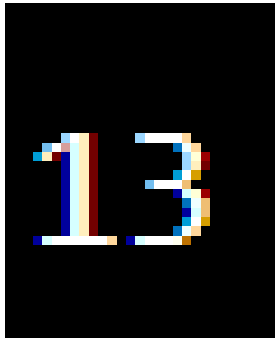
```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

len()

- returns the length of a string
- Example:

```
a = "Hello, World!"
print(len(a))
```

Output:



lower() and upper()

- The lower() method returns the string in lower case
- The upper() method returns the string in upper case:
- Example:

```
a = "Hello, World!"
print(a.lower())
print(a.upper())
```

Output:

```
hello, world!
HELLO, WORLD!
```

replace() and split()

- The replace() method replaces a string with another string
- The split() method splits the string into substrings if it finds instances of the separator
- Example:

```
a = "Hello, World!"
print(a.replace("H", "J"))
print(a.split(",")) # returns ['Hello', ' World!']
```

Output:

```
Jello, World!
['Hello', 'World!']
```

Command-line String Input

- Python allows for command line input.
- That means we are able to ask the user for input.
- The following example asks for the user's name, then, by using the input() method, the program prints the name to the screen

Example: *demo_string_input.py*

```
print("Enter your name:")
x = input()
print("Hello, " + x)
```

Our program will prompt the user for a string:

```
Enter your name:
```

The user now enters a name:

```
Linus
```

Then, the program prints it to screen with a little message:

```
Hello, Linus
```

END

MALAYAN COLLEGES LAGUNA
COLLEGE OF COMPUTER AND INFORMATION SCIENCE

Week 2
Variables, Expressions and Statements

SY 2019-2020 1st Term

Rhea N. Tortor - IT

Constants

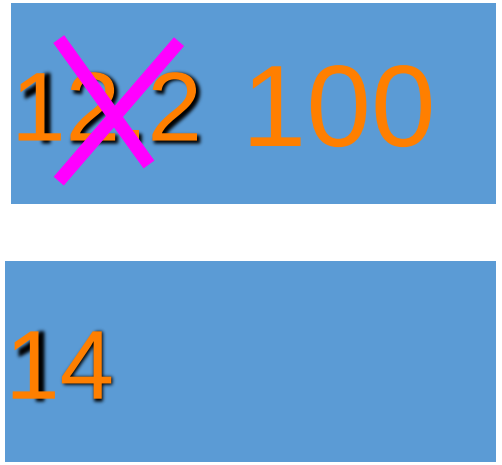
- Fixed values such as numbers, letters, and strings are called "constants" - because their value does not change
- Numeric constants are as you expect
- String constants use single-quotes (') or double-quotes (")

```
>>> print (123)
123
>>> print (98.6)
98.6
>>> print ("Hello world")
Hello world
```

Variables

- A variable is a named place in the memory where a programmer can store data and later retrieve the data using the variable "name"
- Programmers get to choose the names of the variables
- You can change the contents of a variable in a later statement

```
x = 12.2
y = 14
x = 100
```



Python Variable Name Rules

- Must start with a letter or underscore _
- Must consist of letters and numbers and underscores
- Case Sensitive
- Good: spam eggs spam23 _speed
- Bad: 23spam #sign var.12
- Different: spam Spam SPAM

Reserved Words

- You can not use reserved words as variable names / identifiers

```
and   del   for   is   raise
assert   elif   from   lambda   return
break   else   global   not   try
class   except   if   or   while
continue   exec   import   pass   yield
def   finally   in   print
```

Sentences or Lines

x = 2 ⟵ Assignment Statement

x = x + 2 ⟵ Assignment with expression

print (x) ⟵ Print statement

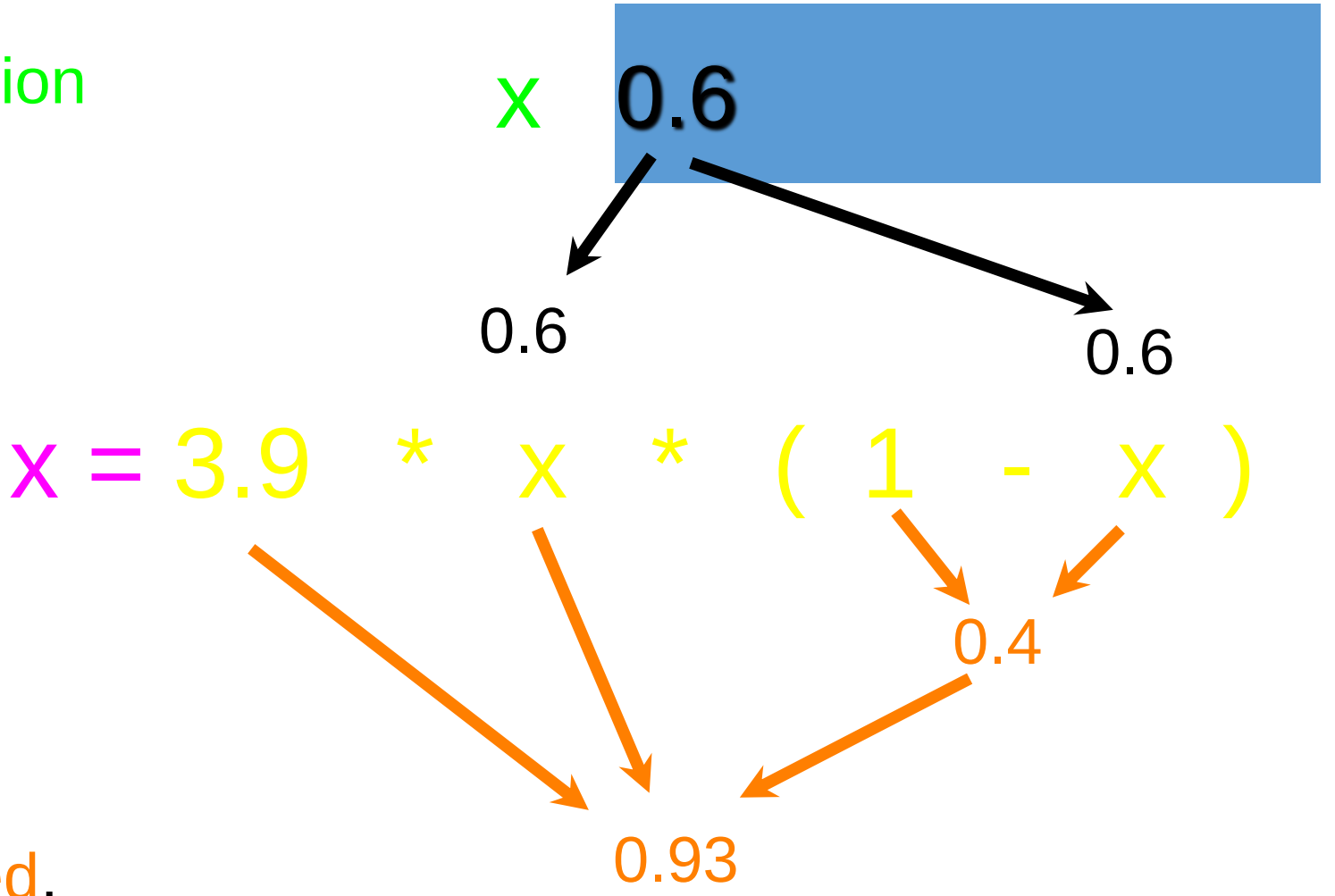
Variable Operator Constant Reserved Word

Assignment Statements

- We assign a value to a variable using the assignment statement (=)
- An assignment statement consists of an expression on the right hand side and a variable to store the result

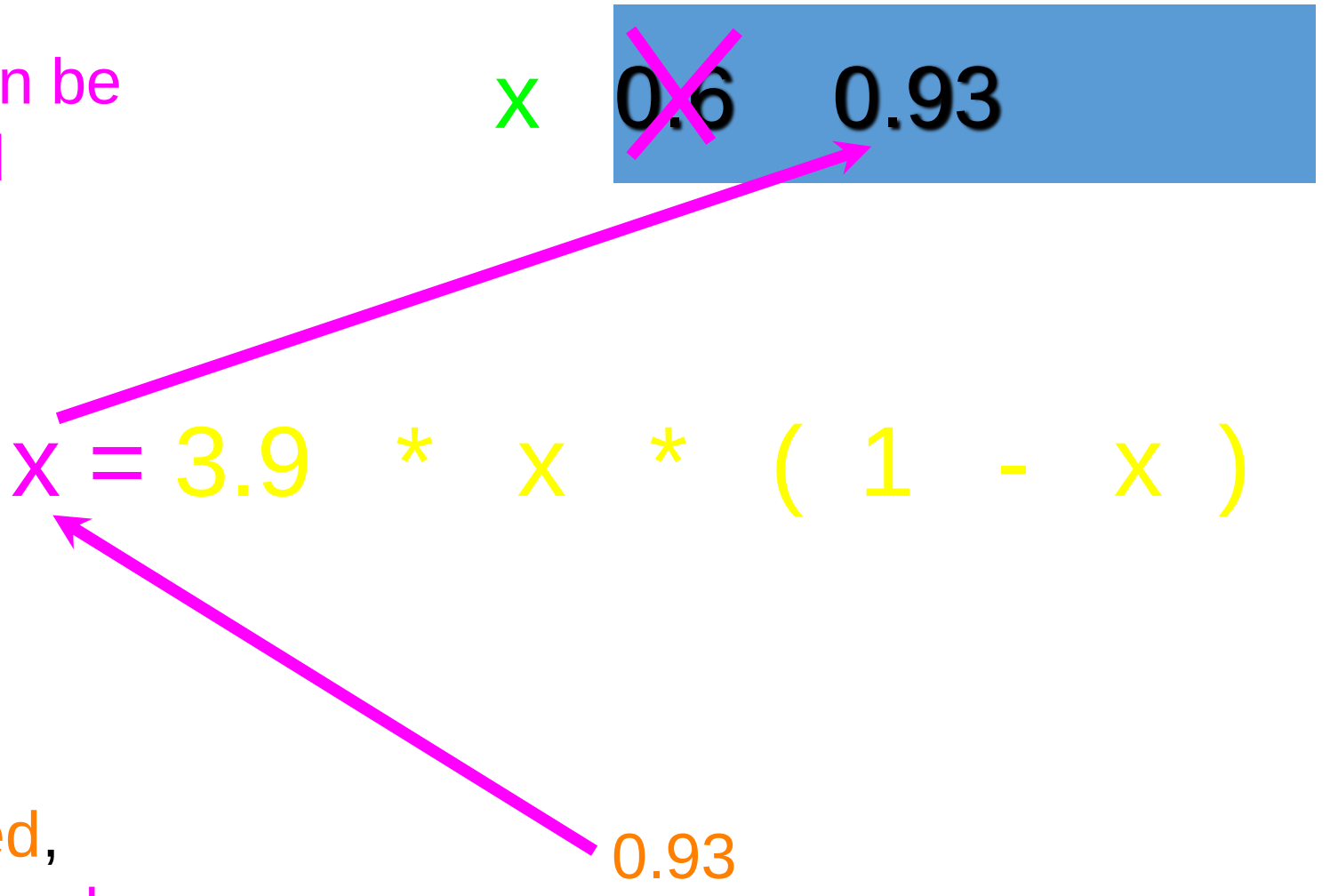
```
x = 3.9 * x * ( 1 - x )
```

A variable is a memory location used to store a value (0.6).



Right side is an expression.
Once expression is evaluated, the result is placed in (assigned to) x.

A variable is a memory location used to store a value. The value stored in a variable can be updated by replacing the old value (0.6) with a new value (0.93).



Right side is an expression.
Once expression is evaluated, the result is placed in (assigned to) the variable on the left side (i.e. x).

Numeric Expressions

- Because of the lack of mathematical symbols on computer keyboards - we use "computer-speak" to express the classic math operations
- Asterisk is multiplication
- Exponentiation (raise to a power) looks different from in math.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

Numeric Expressions

```
>>> xx = 2
>>> xx = xx + 2
>>> print (xx)
4
>>> yy = 440 * 12
>>> print (yy)
5280
>>> zz = yy / 1000
>>> print (zz)
5
```

```
>>> jj = 23
>>> kk = jj % 5
>>> print (kk)
3
>>> print 4 ** 3
64
```

```
    4 R 3
  ┌─────
5 │ 23
    20
   ──
     3
```

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

Order of Evaluation

- When we string operators together - Python must know which one to do first
- This is called "operator precedence"
- Which operator "takes precedence" over the others

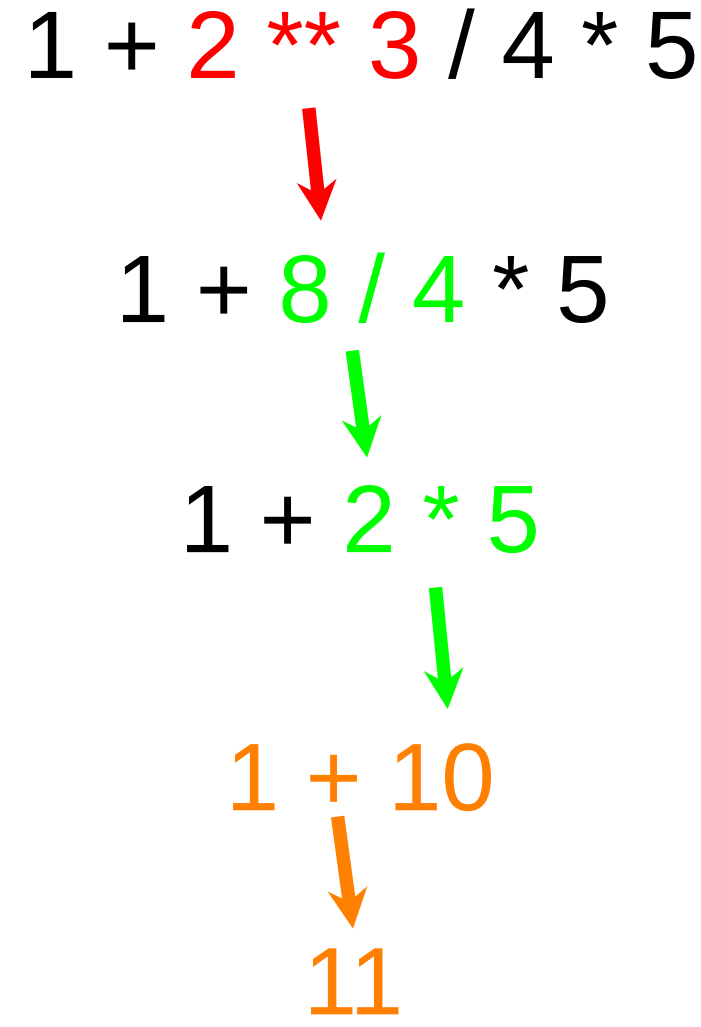
```
x = 1 + 2 * 3 - 4 / 5 ** 6
```

Operator Precedence Rules

- Highest precedence rule to lowest precedence rule
  - Parenthesis are always respected
  - Exponentiation (raise to a power)
  - Multiplication, Division, and Remainder
  - Addition and Subtraction
  - Left to right

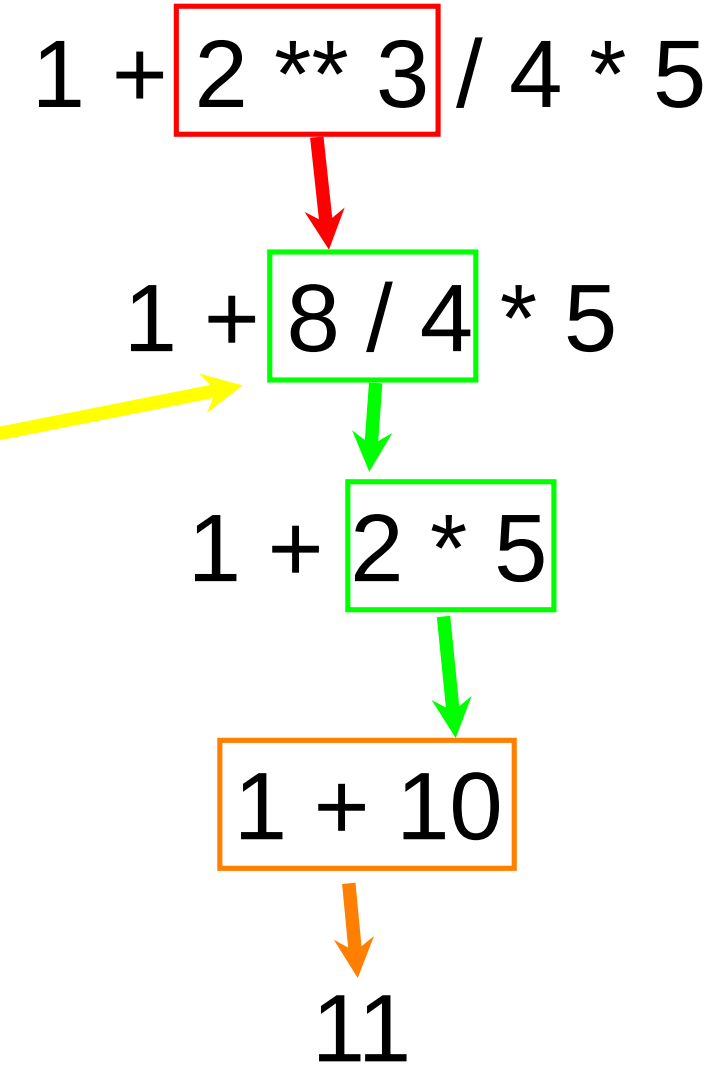
```
>>> x = 1 + 2 ** 3 / 4 * 5
>>> print x
11
>>>
```

Parenthesis
Power
Multiplication
Addition
Left to Right



```
>>> x = 1 + 2 ** 3 / 4 * 5
>>> print x
11
>>>
```

Note 8/4 goes before 4*5 because of the left-right rule.



Parenthesis
Power
Multiplication
Addition
Left to Right

Operator Precedence

- Remember the rules top to bottom
- When writing code - use parenthesis
- When writing code - keep mathematical expressions simple enough that they are easy to understand
- Break long series of mathematical operations up to make them more clear

Exam Question: x = 1 + 2 * 3 - 4 / 5

Python Integer Division is Weird!

- Integer division truncates
- Floating point division produces floating point numbers

```
>>> (print 10 / 2)
5
>>> (print 9 / 2)
4
>>> print (99 / 100)
0
>>> print (10.0 / 2.0)
5.0
>>> print (99.0 / 100.0)
0.99
```

This changes in Python 3.0

Mixing Integer and Floating

- When you perform an operation where one operand is an integer and the other operand is a floating point the result is a floating point
- The integer is converted to a floating point before the operation

```
>>> print (99 / 100)
0
>>> print (99 / 100.0)
0.99
>>> print (99.0 / 100)
0.99
>>> print (1 + 2 * 3 / 4.0 – 5)
-2.5
>>>
```

What does “Type” Mean?

- In Python variables, literals, and constants have a “type”
- Python knows the difference between an integer number and a string
- For example “+” means “addition” if something is a number and “concatenate” if something is a string

```
>>> ddd = 1 + 4
>>> print (ddd)
5
>>> eee = 'hello ' + 'there'
>>> print (eee)
hello there
```

concatenate = put together

Type Matters

- Python knows what "type" everything is
- Some operations are prohibited
- You cannot "add 1" to a string
- We can ask Python what type something is by using the type() function.

```
>>> eee = 'hello ' + 'there'
>>> eee = eee + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> type(eee)
<type 'str'>
>>> type('hello')
<type 'str'>
>>> type(1)
<type 'int'>
>>>
```

Several Types of Numbers

- Numbers have two main types
  - Integers are whole numbers: -14, -2, 0, 1, 100, 401233
  - Floating Point Numbers have decimal parts: -2.5 , 0.0, 98.6, 14.0
- There are other number types - they are variations on float and integer

```
>>> xx = 1
>>> type (xx)
<type 'int'>
>>> temp = 98.6
>>> type(temp)
<type 'float'>
>>> type(1)
<type 'int'>
>>> type(1.0)
<type 'float'>
>>>
```

Type Conversions

- When you put an integer and floating point in an expression the integer is implicitly converted to a float
- You can control this with the built in functions int() and float()

```
>>> print (float(99) / 100)
0.99
>>> i = 42
>>> type(i)
<type 'int'>
>>> f = float(i)
>>> print (f)
42.0
>>> type(f)
<type 'float'>
>>> print (1 + 2 * float(3) / 4 – 5)
-2.5
>>>
```

String Conversions

- You can also use int() and float() to convert between strings and integers
- You will get an **error** if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<type 'str'>
>>> print (sval + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int'
>>> ival = int(sval)
>>> type(ival)
<type 'int'>
>>> print (ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
```

User Input

- We can instruct Python to pause and read data from the user using the input function
- The input function returns a string

```
nam = input( 'Who are you?' )
print ("Welcome", nam)
```

```
Who are you? Chuck
Welcome Chuck
```

Converting User Input

- If we want to read a number from the user, we must convert it from a string to a number using a type conversion function
- Later we will deal with bad input data

```
inp = input( ‘Europe floor?’ )
usf = int(inp) + 1
print ('US floor', usf)
```

```
Europe floor? 0
US floor 1
```

String Operations

- Some operators apply to strings
  - + implies “concatenation”
  - * implies “multiple concatenation”
- Python knows when it is dealing with a string or a number and behaves appropriately

```
>>> print ('abc' + '123')
abc123
>>> print ('Hi' * 5)
HiHiHiHiHi
>>>
```

Summary

- Type
- Reserved words
- Variables (mnemonic)
- Operators
- Operator precedence
- Integer Division
- Conversion between types
- User input

MALAYAN COLLEGES LAGUNA
COLLEGE OF COMPUTER AND INFORMATION SCIENCE

Week 3
Boolean Expressions

SY 2019-2020 1st Term

Rhea N. Tortor - IT

Topics

- The `if` Statement
- The `if-else` Statement
- Logical Operators
- Boolean Variables

The `if` Statement

- Control structure: logical design that controls order in which set of statements execute
- Sequence structure: set of statements that execute in the order they appear
- Decision structure: specific action(s) performed only if a condition exists
  - Also known as selection structure

The `if` Statement (cont'd.)

- In flowchart, diamond represents true/false condition that must be tested
- Actions can be *conditionally executed*
  - Performed only when a condition is true
- Single alternative decision structure: provides only one alternative path of execution
  - If condition is not true, exit the structure

The `if` Statement (cont'd.)

- **Python syntax:**

```
if condition:
    Statement
    Statement
```

- First line know as the `if` clause
  - Includes the keyword `if` followed by condition
    - The condition can be true or false
    - When the `if` statement executes, the condition is tested, and if it is true the block statements are executed. otherwise, block statements are skipped

Boolean Expressions and Relational Operators

- Boolean expression: expression tested by if statement to determine if it is true or false
  - Example: a > b
    - `true` if a is greater than b; `false` otherwise
- Relational operator: determines whether a specific relationship exists between two values
  - Example: greater than (>)

Boolean Expressions and Relational Operators (cont'd.)

- >= and <= operators test more than one relationship
  - It is enough for one of the relationships to exist for the expression to be true
- == operator determines whether the two operands are equal to one another
  - Do not confuse with assignment operator (=)
- != operator determines whether the two operands are not equal

Boolean Expressions and Relational Operators (cont'd.)

Table 4-2 Boolean expressions using relational operators

Expression	Meaning
`x > y`	Is x greater than y?
`x < y`	Is x less than y?
`x >= y`	Is x greater than or equal to y?
`x <= y`	Is x less than or equal to y?
`x == y`	Is x equal to y?
`x != y`	Is x not equal to y?

Boolean Expressions and Relational Operators (cont'd.)

- Any relational operator can be used in a decision block
  - Example: `if balance == 0`
  - Example: `if payment != balance`
- It is possible to have a block inside another block
  - Example: `if` statement inside a function
  - Statements in inner block must be indented with respect to the outer block

The `if-else` Statement

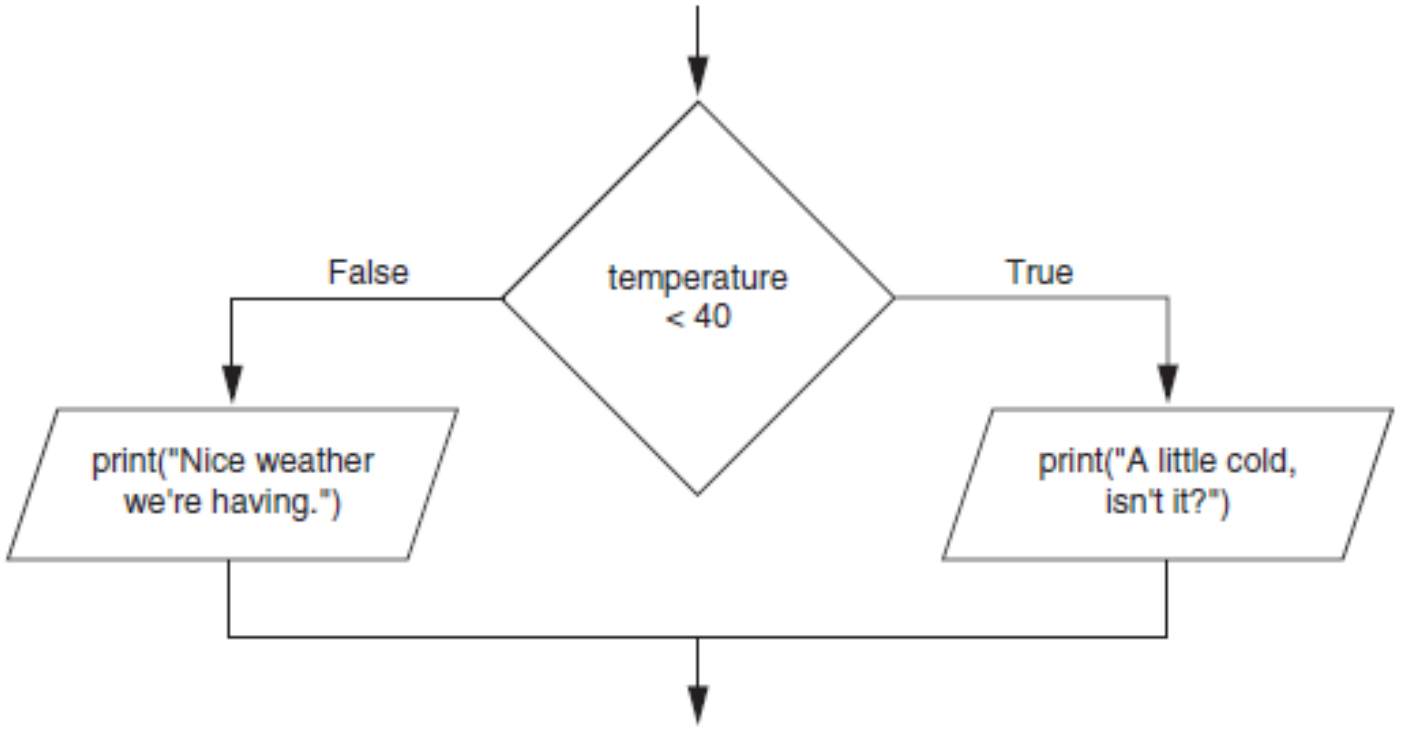
- **Dual alternative decision structure: two possible paths of execution**
  - **One is taken if the condition is true, and the other if the condition is false**
  - **Syntax:**

```
if condition:
        statements
else:
        other statements
```

  - `if` clause and `else` clause must be aligned
  - Statements must be consistently indented

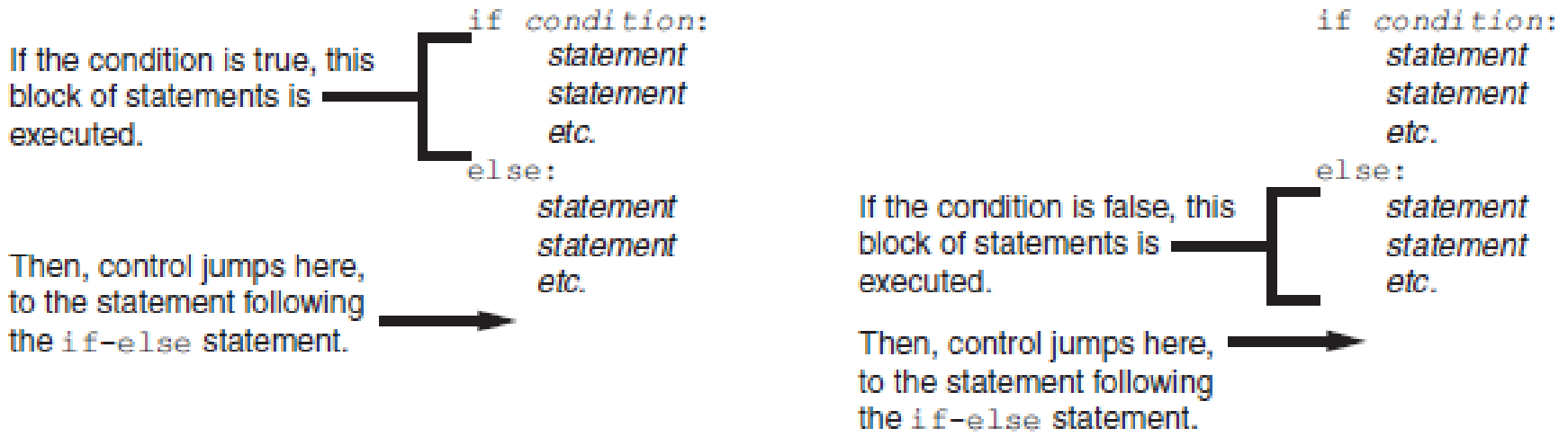
The `if-else` Statement (cont'd.)

Figure 4-6 A dual alternative decision structure



The `if-else` Statement (cont'd.)

Figure 4-7 Conditional execution in an `if-else` statement



Logical Operators

- Logical operators: operators that can be used to create complex Boolean expressions
  - `and` operator and `or` operator: binary operators, connect two Boolean expressions into a compound Boolean expression
  - `not` operator: unary operator, reverses the truth of its Boolean operand

The and Operator

- Takes two Boolean expressions as operands
  - Creates compound Boolean expression that is true only when both sub expressions are true
  - Can be used to simplify nested decision structures
- Truth table for the `and` operator

Expression	Value of the Expression
false and false	false
false and true	false
true and false	false
true and true	true

The `or` Operator

- Takes two Boolean expressions as operands
  - Creates compound Boolean expression that is true when either of the sub expressions is true
  - Can be used to simplify nested decision structures
- Truth table for the `or` operator

Expression	Value of the Expression
false or false	false
false or true	true
true or false	true
true or true	true

The `not` Operator

- Takes one Boolean expressions as operand and reverses its logical value
  - Sometimes it may be necessary to place parentheses around an expression to clarify to what you are applying the not operator
- Truth table for the `not` operator

Expression	Value of the Expression
true	false
false	true

Checking Numeric Ranges with Logical Operators

- To determine whether a numeric value is within a specific range of values, use `and`
  - Example: `x >= 10 and x <= 20`
- **To determine whether a numeric value is outside of a specific range of values, use** `or`
  - Example: `x < 10 or x > 20`

Boolean Variables

- **Boolean variable: references one of two values, `True` or `False`**
  - **Represented by `bool` data type**
- Commonly used as flags
  - Flag: variable that signals when some condition exists in a program
    - Flag set to `False` → condition does not exist
    - Flag set to `True` → condition exists

Summary

- This chapter covered:
  - Decision structures, including:
    - Single alternative decision structures
    - Dual alternative decision structures
    - Nested decision structures
  - Relational operators and logical operators as used in creating Boolean expressions
  - String comparison as used in creating Boolean expressions
  - Boolean variables

END

MALAYAN COLLEGES LAGUNA
COLLEGE OF COMPUTER AND INFORMATION SCIENCE

Week 4
Iterative Control

SY 2019-2020 1st Term

Rhea N. Tortor - IT

Contents

- WHILE Statement
- Infinite Loops
- Definite VS. Indefinite Loops

Definition of Terms

- **accumulator:** A variable used in a loop to add up or accumulate a result.
- **counter:** A variable used in a loop to count the number of times something happened. We initialize a counter to zero and then increment the counter each time we want to "count" something.
- **decrement:** An update that decreases the value of a variable.

Definition of Terms

- **initialize:** An assignment that gives an initial value to a variable that will be updated.
- **increment:** An update that increases the value of a variable (often by one).
- **infinite loop:** A loop in which the terminating condition is never satisfied or for which there is no termination condition.
- **iteration:** Repeated execution of a set of statements using either a recursive function call or a loop.

The while statement

- Repetition Structures
  - Allow a program to repeat an action while a condition is true
- Using `while` Repetition
  - Action(s) contained within the *body* of the loop
  - *Condition* should evaluate to false at some point
    - Otherwise infinite loop and program hangs

`while` Repetition Structure

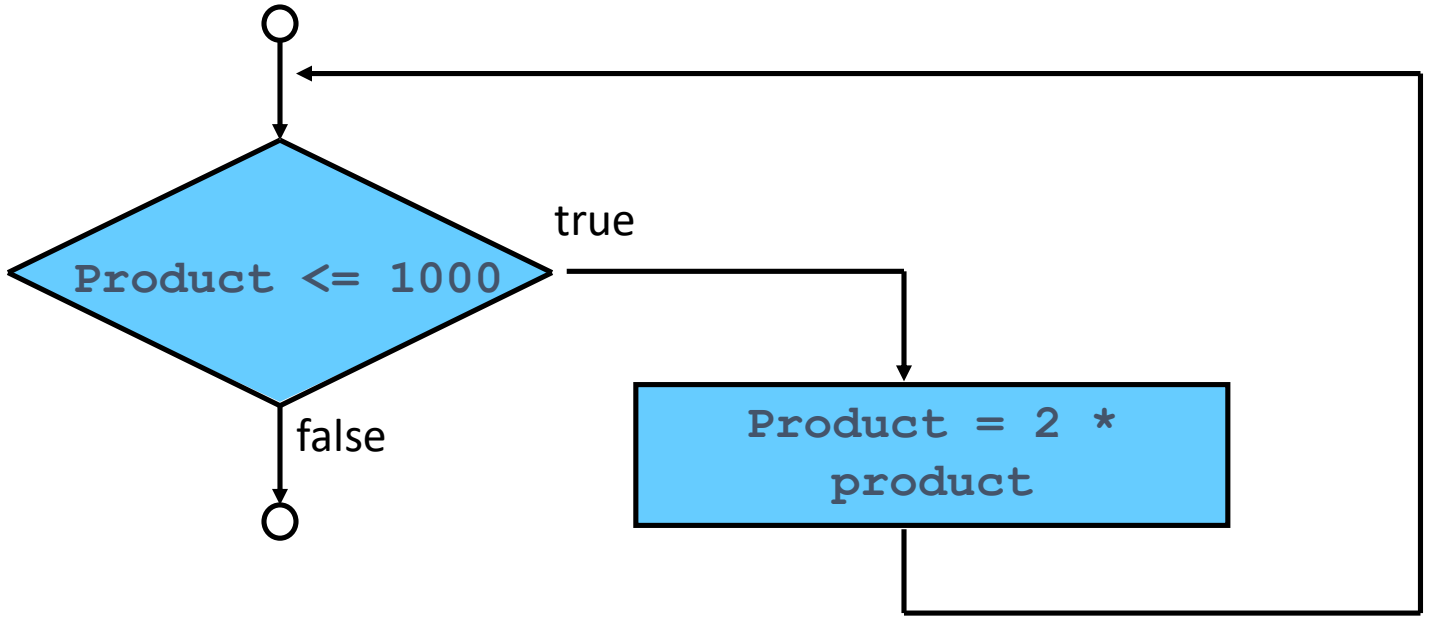


Fig. 3.8 **`while`** repetition structure flowchart.

Example

- Here is a simple program that counts down from five and then says "Blastoff!".

```
n = 5
while n > 0:
    print n
    n = n-1
print (“Blastoff!”)
```

You can almost read the while statement as if it were English. It means, "While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, exit the while statement and display the word Blastoff!"

More formally, here is the flow of execution for a *while* statement:

1. Evaluate the condition, yielding True or False.
2. If the condition is false, exit the while statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

- This type of flow is called a *loop* because the third step loops back around to the top.
- Each time we execute the body of the loop, we call it an *iteration*.
- The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates.
- We call the variable that changes each time the loop executes and controls when the loop finishes the *iteration variable*.
- If there is no iteration variable, the loop will repeat forever, resulting in an *infinite loop*.

Breaking Out of a Loop

- The break statement ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop

```
while True:
    line = raw_input('> ')
    if line == 'done' :
        break
    print (line)
print ('Done!‘)
```

```
> hello there
hello there
> finished
finished
> done
Done!
```

Finishing an Iteration with continue

- The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print (line)
print ('Done!‘)
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

Indefinite Loops

- While loops are called "indefinite loops" because they keep going until a logical condition becomes False
- The loops we have seen so far are pretty easy to examine to see if they will terminate or if they will be "infinite loops"
- Sometimes it is a little harder to be sure if a loop will terminate

Definite Loops

- Quite often we have a list of items of the lines in a file - effectively a finite set of things
- We can write a loop to run the loop once for each of the items in a set using the Python for construct
- These loops are called "definite loops" because they execute an exact number of times
- We say that "definite loops iterate through the members of a set"

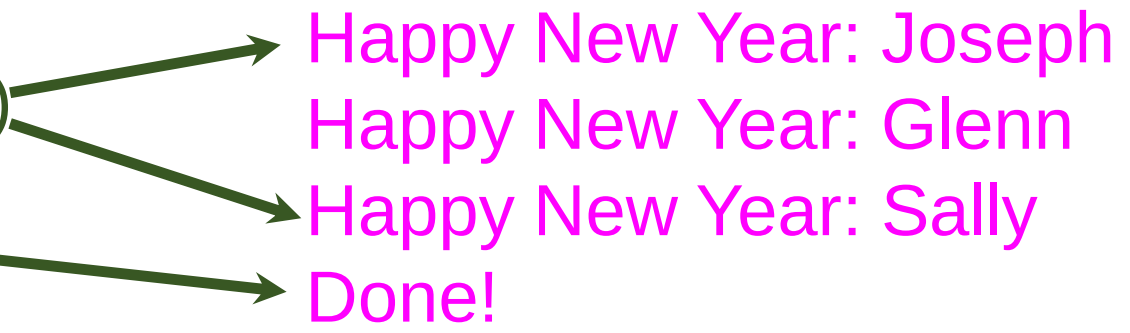
A Simple Definite Loop

```
for i in [5, 4, 3, 2, 1] :
    print (i)
print ('Blastoff!‘)
```

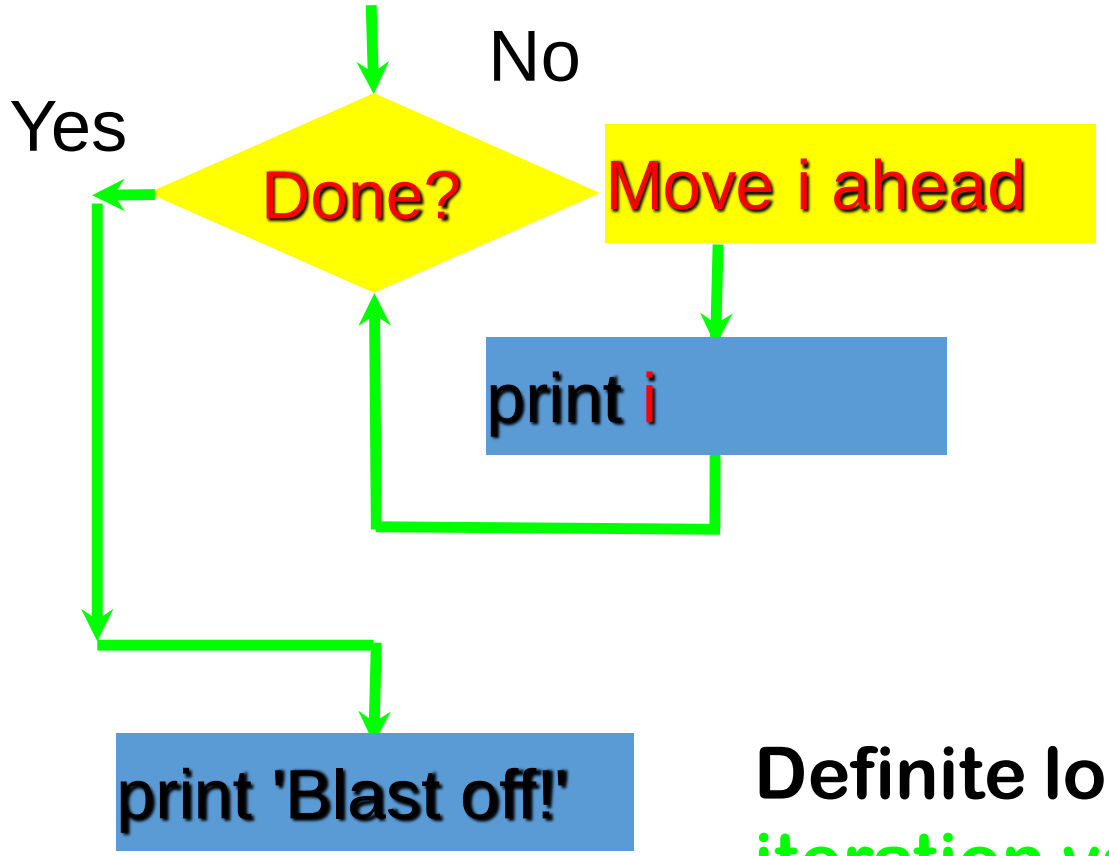
```
5
4
3
2
1
Blastoff!
```

A Definite Loop with Strings

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends :
    print ('Happy New Year:',  friend)
print ('Done!')
```



A Simple Definite Loop



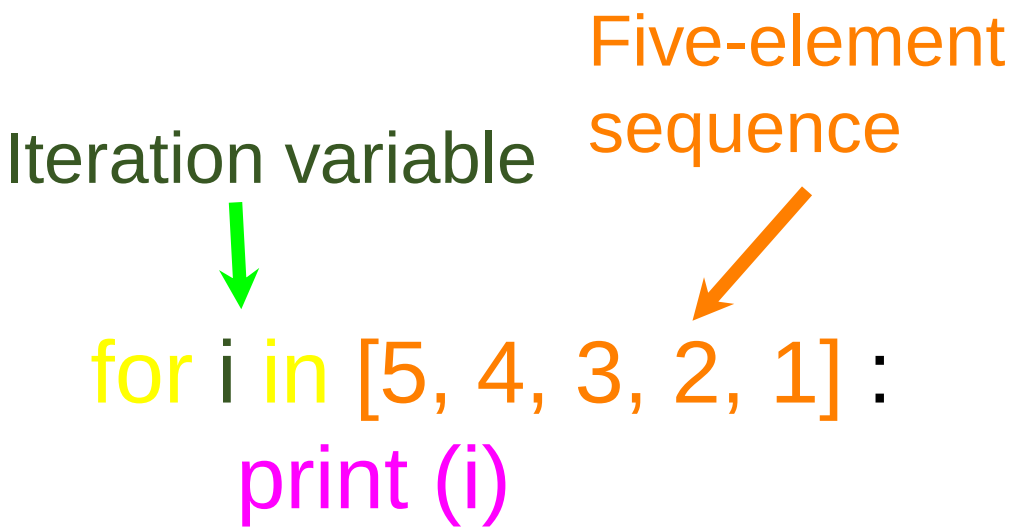

```
for i in [5, 4, 3, 2, 1] :
    print (i)
print ('Blastoff!‘)
```

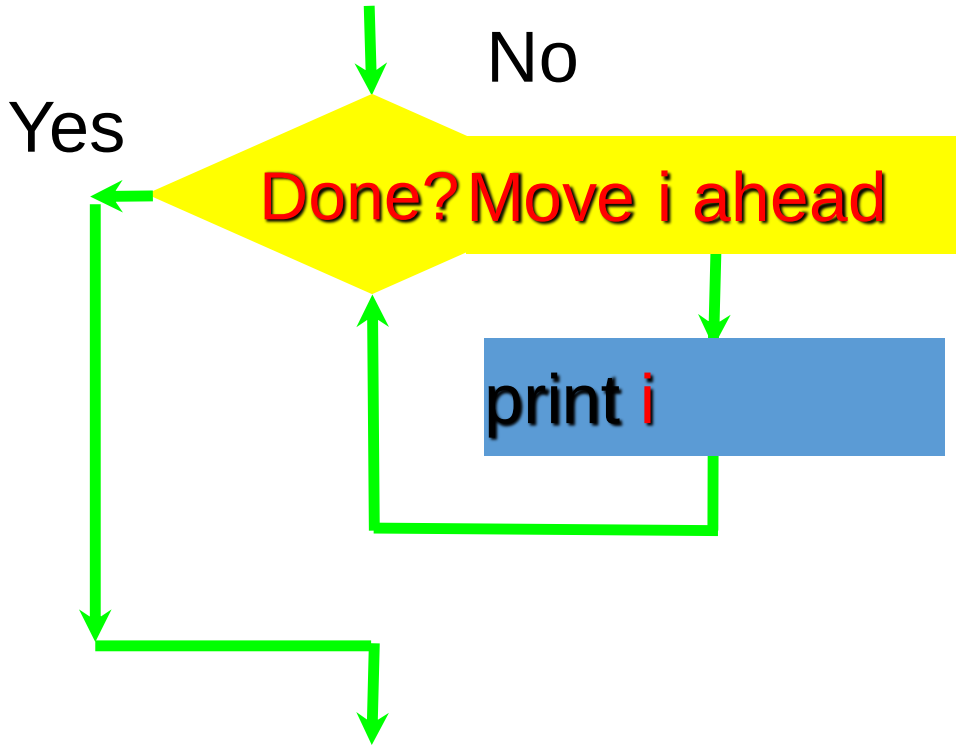
```
5
4
3
2
1
Blastoff!
```

Definite loops (for loops) have explicit iteration variables that change each time through a loop. These iteration variables move through the sequence or set.

Looking at In...

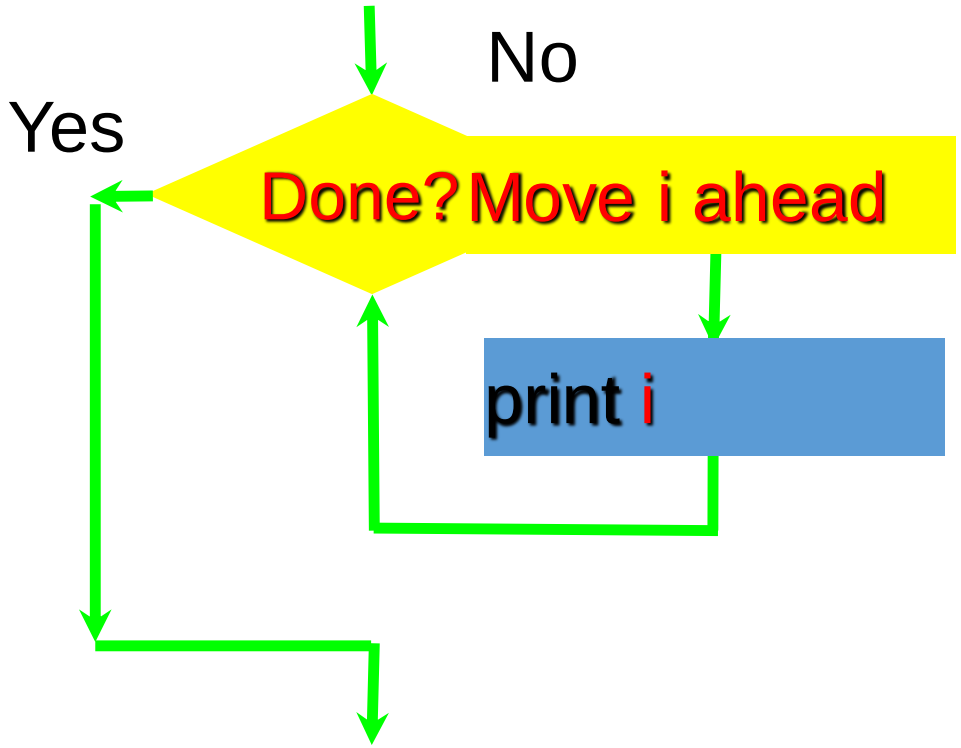
- The iteration variable "iterates" though the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence



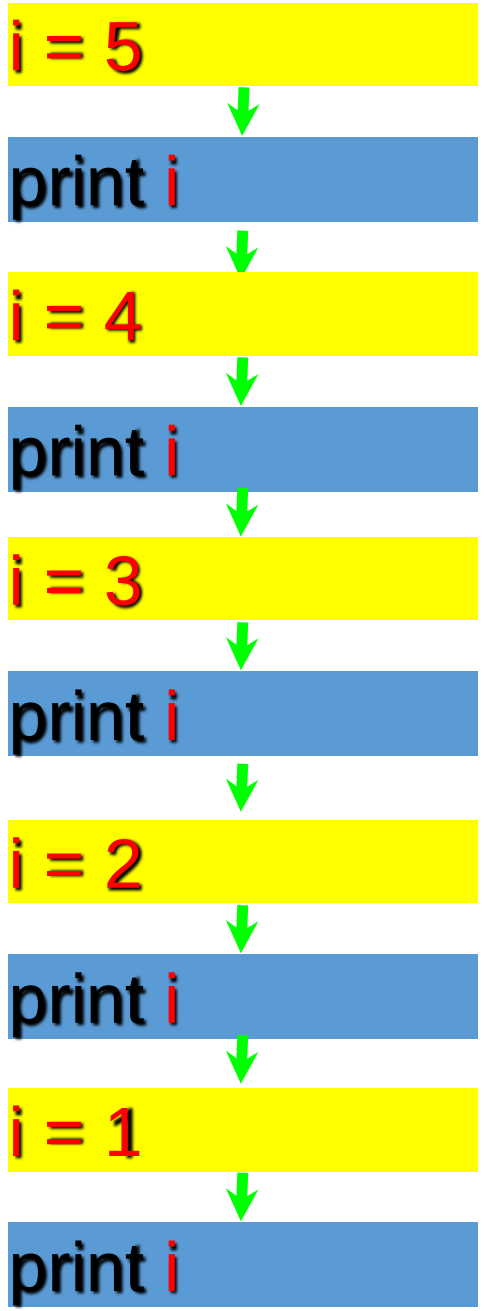



```
for i in [5, 4, 3, 2, 1]:
    print (i)
```

- The iteration variable "iterates" though the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence




```
for i in [5, 4, 3, 2, 1] :
    print (i)
```



Definite Loops

- Quite often we have a list of items of the lines in a file - effectively a finite set of things
- We can write a loop to run the loop once for each of the items in a set using the Python for construct
- These loops are called "definite loops" because they execute an exact number of times
- We say that "definite loops iterate through the members of a set"

Counting in a Loop

```
zork = 0
print ('Before', zork)
for thing in [9, 41, 12, 3, 74, 15] :
    zork = zork + 1
    print (zork, thing)
print ('After', zork)
```

```
$ python countloop.py
Before 0
1 9
2 41
3 12
4 3
5 74
6 15
After 6
```

To count how many times we execute a loop we introduce a counter variable that starts at 0 and we add one to it each time through the loop.

Summing in a Loop

```
zork = 0
print ('Before', zork)
for thing in [9, 41, 12, 3, 74, 15] :
    zork = zork + thing
    print (zork, thing)
print ('After', zork)
```

```
$ python countloop.py
Before 0
9 9
50 41
62 12
65 3
139 74
154 15
After 154
```

To add up a value we encounter in a loop, we introduce a sum variable that starts at 0 and we add the value to the sum each time through the loop.

Finding the Average in a Loop

```
count = 0
sum = 0
print ('Before', count, sum)
for value in [9, 41, 12, 3, 74, 15] :
    count = count + 1
    sum = sum + value
    print (count, sum, value)
print ('After', count, sum, sum / /count)
```

```
$ python averageloop.py
Before 0 0
1 9 9
2 50 41
3 62 12
4 65 3
5 139 74
6 154 15
After 6 154 25
```

An average just combines the counting and sum patterns and divides when the loop is done.

Filtering in a Loop

```
print (‘Before’)
for value in [9, 41, 12, 3, 74, 15] :
    if value > 20:
            print ('Large number',value)
print (‘After’)
```

```
$ python search1.py
Before
Large number 41
Large number 74
After
```

We use an if statement in the loop to catch / filter the values we are looking for.

Search Using a Boolean Variable

```
found = False
print ('Before', found)
for value in [9, 41, 12, 3, 74, 15] :
    if value == 3 :
        found = True
    print (found, value)
print ('After', found)
```

```
$ python search1.py
Before False
False 9
False 41
False 12
True 3
True 74
True 15
After True
```

If we just want to search and know if a value was found - we use a variable that starts at False and is set to True as soon as we find what we are looking for.

The "is" and "is not" Operators

```
smallest = None
print 'Before'
for value in [3, 41, 12, 9, 74, 15] :
    if smallest is None :
        smallest = value
    elif value < smallest :
        smallest = value
    print smallest, value
print 'After', smallest
```

- Python has an "is" operaror that can be used in logical expressions
- Implies 'is the same as'
- Similar to, but stronger than ==
- 'is not' also is a logical operator

Summary

- While loops (indefinite)
- Infinite loops
- Using break
- Using continue
- For loops (definite)
- Iteration variables