# Module1

## Python  Programming Basics

## Syllabus: Module 1

**Python Basics**: Entering Expressions into the Interactive Shell, The Integer, Floating-Point, and String Data Types, String Concatenation and Replication, Storing Values in Variables, Your First Program, Dissecting Your Program

**Flow control:** Boolean Values, Comparison Operators, Boolean Operators, Mixing Boolean and Comparison Operators, Elements of Flow Control, Program Execution, Flow Control Statements, Importing Modules, Ending a Program Early with sys.exit(),

**Functions:** def Statements with Parameters, Return Values and return Statements, The None Value, Keyword Arguments and print(), Local and Global Scope, The global Statement, Exception Handling, A Short Program: Guess the Number

**Text Book-"Automate the Boring Stuff with Python" by** Al Sweigart **:Chapters 1 and 2.**

## Python Basics: Expressions

```
>>> 7+2       # Addition
9            #Output
>>> 7-2       # Subtraction
5
>>> 7*2       #Multiplication
14
>>> 7/2       #Division
3.5
>>> 7**2      #Exponentiation
49
```

# Python Basics: **Expressions**

```
>>> 7/2    #Division    (real (floating) division)
3.5
>>> 7//2    #Integer Division. (the integer part of the real division)
3
>>> 7%2
1
```

What are the Quotient, remainder in Python?

- When two numbers divide with each other, the result is **quotient**.
- Divide two numbers using **'//'** operator to get the quotient.
- The **remainder** is calculated using the **'%'** operator in Python.

# Python Basics: **Expressions**

- Expression: is the most basic kind of programming instruction in Python.
- Expressions consist of values (Ex:2) and operators (Ex: +), and they can always evaluate (= reduce) down to a single value.
- Expressions can be used anywhere in Python code where a value could be used.
- A single value with no operators is also considered an expression, though it evaluates only to itself.
- Ex:

```
>>> 2
2
```

# Python Basics: **Expressions**

- Precedence of operators: The order of operations of Python math operators
- It is similar to that of mathematics. From left to right

1. **
2. *, /, //, and %
3. + and -

- Use parentheses to override the usual precedence if needed.
- Whitespace in between the operators and values doesn't matter for Python (except for the indentation at the beginning of the line)

# Python Basics: **Expressions**

- From left to right

  1. **    2. *, /, //, and %    3. + and -

- Use parentheses to override the usual precedence if needed.

```
>>> 2+3*6
20
>>> (2+3)*6
30
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

- Python will keep evaluating parts of the expression until it becomes a single value

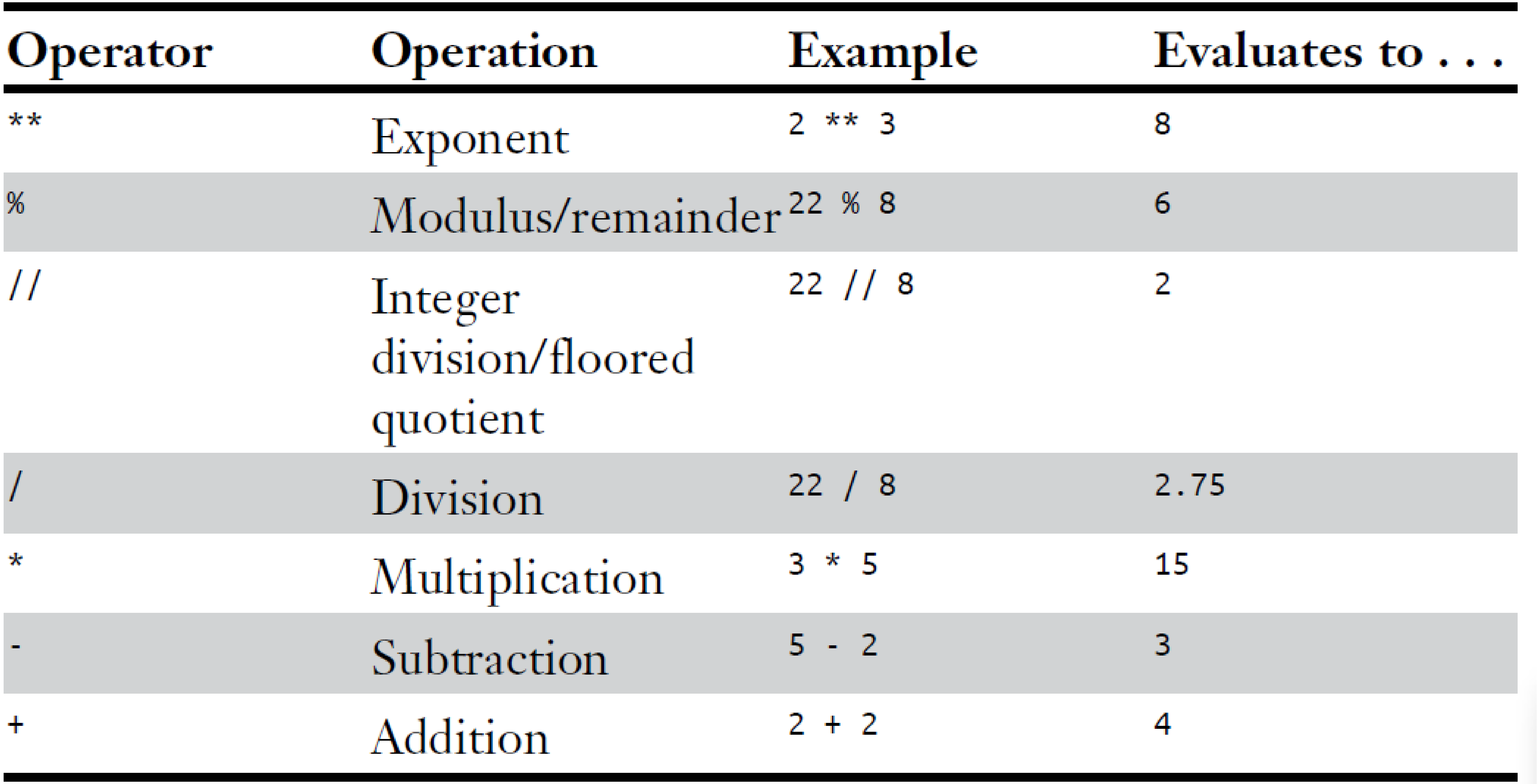
# Python Basics: **Expressions**

- From left to right

  1. **  2. *, /, //, and %  3. + and -

```
>>> (5 - 1) * ((7 + 1) / (3 - 1))
4 * ((7 + 1) / (3 - 1))
4 * (8 / (3 - 1))
4 * (8 / 2)
4 * 4.0
16.0
```

# Python Basics: **Expressions**

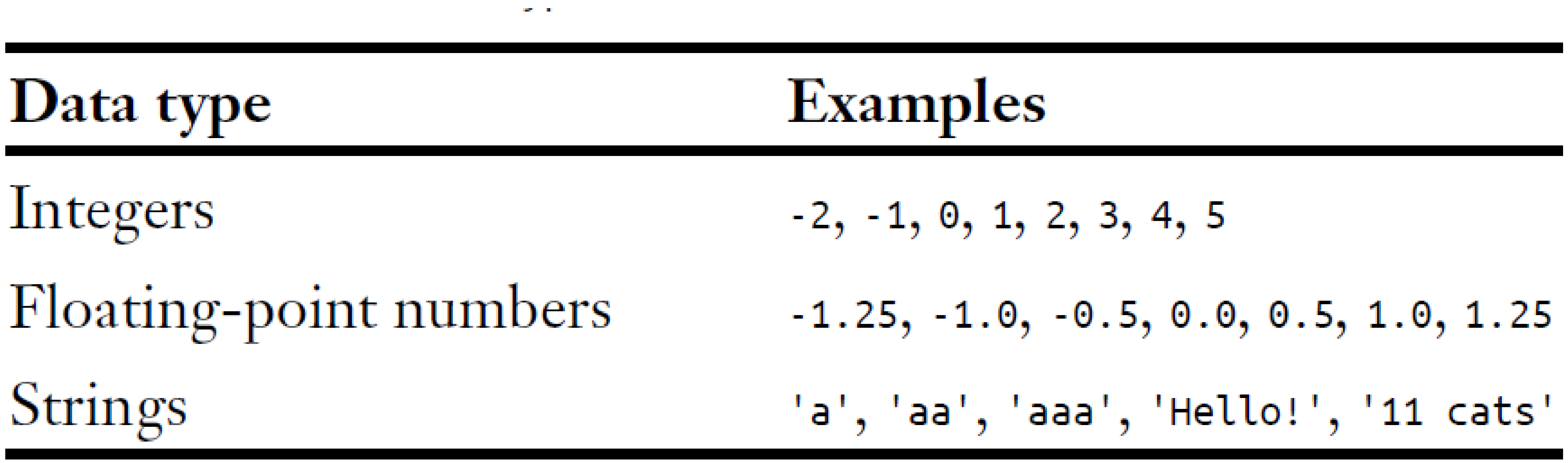


| Operator | Operation | Example | Evaluates to . . . |
|---|---|---|---|
| `**` | Exponent | `2 ** 3` | `8` |
| `%` | Modulus/remainder | `22 % 8` | `6` |
| `//` | Integer division/floored quotient | `22 // 8` | `2` |
| `/` | Division | `22 / 8` | `2.75` |
| `*` | Multiplication | `3 * 5` | `15` |
| `-` | Subtraction | `5 - 2` | `3` |
| `+` | Addition | `2 + 2` | `4` |

## The Integer, Floating-Point, and String Data Types

- A *data type* is a category for values, and
- Every value belongs to exactly one data type

Most common data types:



| Data type | Examples |
|---|---|
| Integers | `-2, -1, 0, 1, 2, 3, 4, 5` |
| Floating-point numbers | `-1.25, -1.0, -0.5, 0.0, 0.5, 1.0, 1.25` |
| Strings | `'a', 'aa', 'aaa', 'Hello!', '11 cats'` |

## The Integer, Floating-Point, and String Data Types

**Integers :**

- An integer is a whole number that can be positive or negative.
- Ex: `10` `-10` `123456789`
- There is no limit to how long an integer value can be. It can grow to have as many digits as computer's memory space allows.
- Ex: 99999999999999999999999999999999999999999999999999
- Normally integers are written in base 10. However, Python allows to write integers in Hexadecimal (base 16), Octal (base 8), and Binary (base 2) formats.

## The Integer, Floating-Point, and String Data Types

**Floating-Point Number(Float):**

- Float is a positive or negative number with a fractional part(decimal point).
- Ex: 10.1   -10.5   1.123456

-
  ```
  >>> type(42.0)
  <class 'float'>
  ```

-
  ```
  >>> type(42)
  <class 'int'>
  ```

## The Integer, Floating-Point, and String Data Types

**Strings:**

- Text values used in Python program are called strings or strs
- Always surround string in single quote or double quote or triple quote.
- Ex: 'Hello World‘, “Hello World”, ‘”Hello World”‘

```
>>> 'Hello World'
'Hello World'
>>> "Hello World"
'Hello World'
>>> "'Hello World'"
"'Hello World'"
```

# The Integer, Floating-Point, and String Data Types

**Strings:**

- String with no characters in it - *blank string* or an *empty string*
- **If the error message is :**

**SyntaxError: EOL while scanning string literal.**

**Meaning: forgot the final single quote character at the end of the string.**

**Ex: 'Hello World!**

**SyntaxError: EOL while scanning string literal**

# String Concatenation and Replication

- **The meaning of an operator may change based on the data types of the values next to it.**
- **Ex: + is the addition operator when it operates on two integers or floats.**

```
>>> 7+2                          # addition
9                               #Output
```

- **However, when + is used on two string values, it joins the strings as the string concatenation operator.**

```
>>> 'Alice' + 'Bob‘                 # Concatenation
'AliceBob‘
```

- The expression evaluates down to a single, new string value that

 combines the text of the two strings.

# String Concatenation and Replication

```
>>> 'Alice' + 'Bob'
'AliceBob‘
```

- The expression evaluates down to a single, new string value that combines the text of the two strings.
- If + operator is used on a string and an integer value:

Python will not know how to handle this, and it will display an error message.

```
>>> 'Alice'+42
Traceback (most recent call last):
File "<pyshell#0>", line 1, in <module>
'Alice'+42
TypeError: can only concatenate str (not "int") to str
```

# String Concatenation and Replication

- **The * operator multiplies two integer or floating-point values.**
- **When the * operator is used on one string value and one integer value:**

  **It becomes the string replication operator.**
- **Enter a string multiplied by a number into the interactive shell:**

```
>>> 'Alice‘ * 3
'AliceAliceAlice'
```

## String Concatenation and Replication

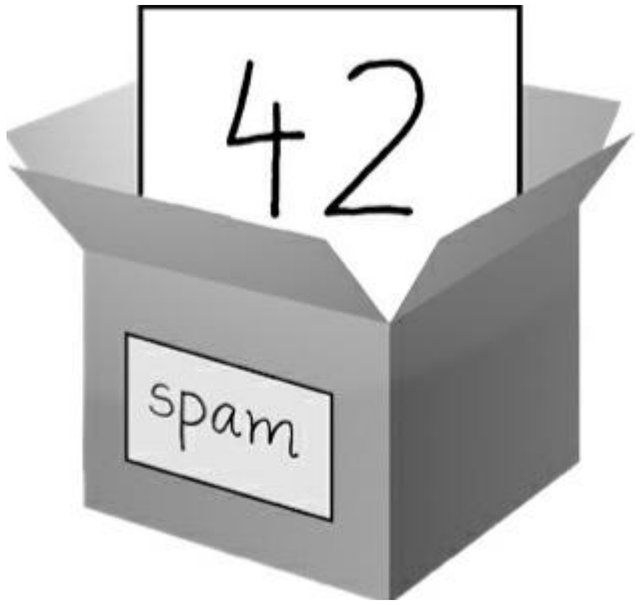
**The * operator can be used with**

- **only two numeric values : for multiplication**
- **one string and one integer : for string replication.**
- **Otherwise:???????**
- **Python will just display an error message:**

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
```

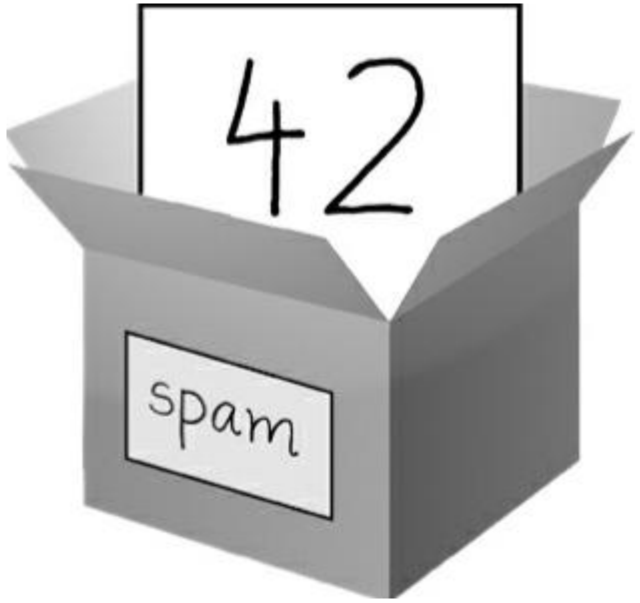
# Storing Values in Variables

- **A variable is like a box in the computer's memory which can be used to store a single value.**
- **The result of an evaluated expression can be saved inside a variable.**
- **Values are stored in variables with an assignment statement.**



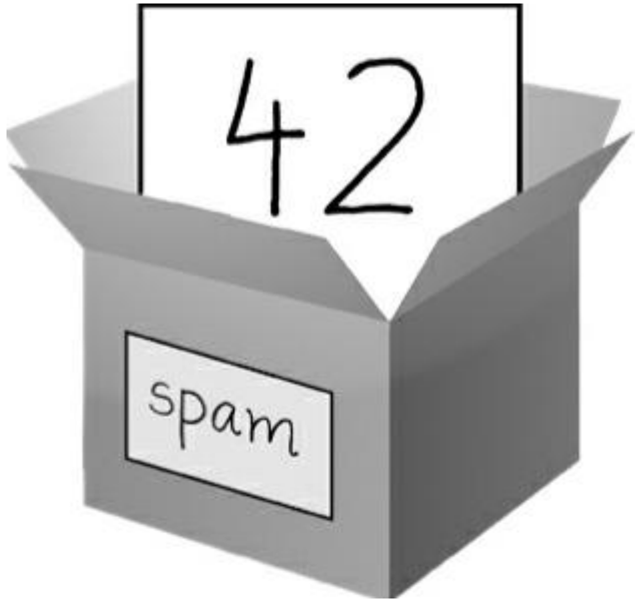
# Storing Values in Variables: Assignment statement

- **An assignment statement consists of :**
  **a variable name,**
  **an equal sign (called assignment operator)**
  **the value to be stored.**
  **Ex: spam = 42**
- **A variable named spam will have the integer value 42 stored in it.**
- **A variable = a labeled box with a value**
- **placed in**



# Storing Values in Variables: Assignment statement

```
❶ >>> spam = 40
>>> spam
40
>>> eggs = 2
❷ >>> spam + eggs
42
>>> spam + eggs + spam
82
❸ >>> spam = spam + 2
>>> spam
42
```



# Storing Values in Variables: Assignment statement

- A variable is *initialized* (or created) the first time a value is stored in it.

❶ >>> **spam = 40;   eggs=2**

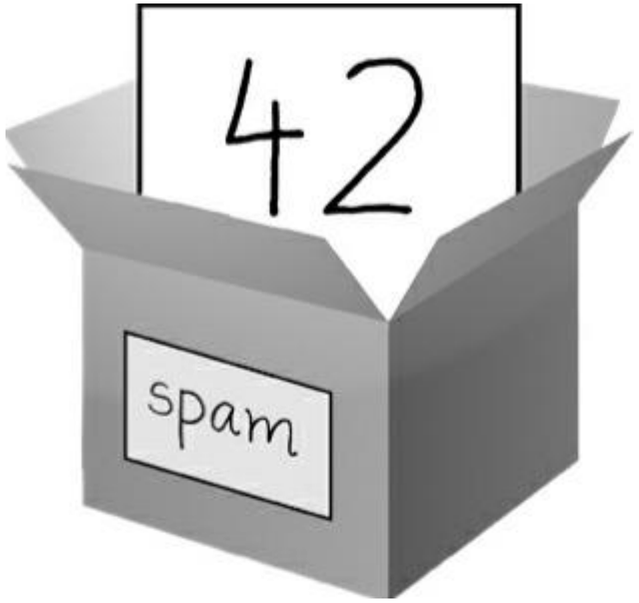
- After that, it can be used in expressions

with other variables and values

❷ >>> **spam + eggs      O/P: 40+2=42**

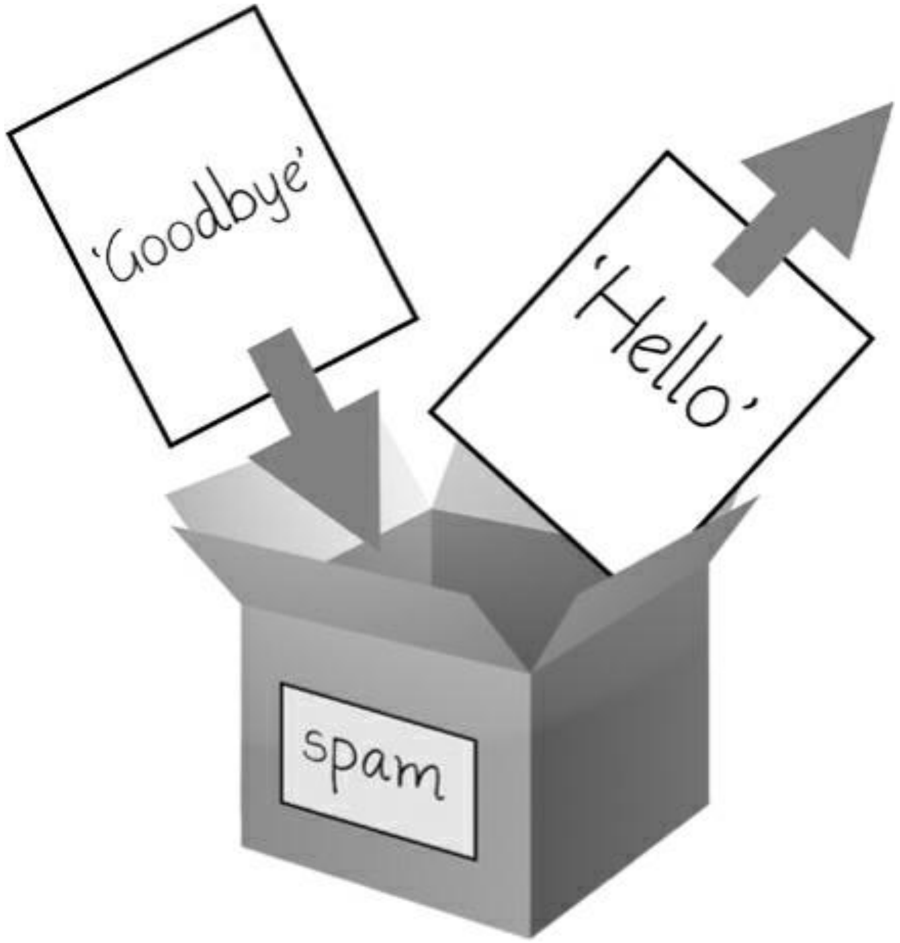
*Overwriting* the variable: When a variable is assigned a new value, the old value is forgotten.

❸ >>> **spam = spam + 2   O/P: 42**



## Storing Values in Variables: Assignment statement

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```



## Variable Names

**Rules to be followed while naming a variable:**

- **It can be only one word with no spaces.**
- **It can use only letters, numbers, and the underscore (_) character.**
- **It can't begin with a number.**

- **Variable names are case-sensitive: spam, SPAM, Spam, and sPaM are four different variables.**
- **Usual Python convention :start variables with a lowercase letter.**

# Variable Names

- A good variable name describes the data it contains.
- Imagine that you moved to a new house and labeled all of the moving boxes as Stuff.
- One would never find anything!
- Descriptive name will help make code more readable.

# Variable Names

| Valid variable names | Invalid variable names |
|---|---|
| current_balance | current-balance |
| currentBalance | current balance |
| account4 | 4account |
| _42 | 42 |
| TOTAL_SUM | TOTAL_SUM |
| hello | 'hello' |

## Your First Program

- **Interactive shell : runs Python instructions one at a time**
- **File editor: To write entire Python program.**

- **The file editor is similar to text editors such as Notepad but it has some special features for entering source code.**
- **To open a new file: click the New button on the top row.**

## Your First Program

**Difference between :**

| **Interactive shell window** | **File editor window** |
|---|---|
| **Has >>> prompt** | **No  >>> prompt** |
| **Runs Python instructions as soon as you press ENTER** | **Lets to type in many instructions, save the file, and run the program.** |
| Runs Python instructions one at a time | Allows to create or edit from existing python files |

# Your First Program

- **How to save the file?**
- **Once you've entered your source code, Click the Save button in the top row,**

  **give the File Name as name.py and then click Save.**
- **.py is the extension of the Python file.**
- **Save program every once in a while as you type them.**
- **To save : press CTRL-S on Windows**
- **To run : Press F5 key**
- **Program runs in the interactive shell window. Press F5 from the file editor window, not the interactive shell window.**

## Your First Program

- **When there are no more lines of code to execute, the Python program terminates; that is, it stops running. (= Python program exits.)**
- **Close the file editor by clicking the X at the top of the window.**
- **Reload a saved program: select File ▸ Open... from the menu.**
- **In the window that appears, choose filename hello.py and click the Open button. Previously saved hello.py program opens in the file editor window.**

## Your First Program

- **View the execution of a program using the Python Tutor visualization tool :**
- **http://pythontutor.com/**
- **You can see the execution of this particular program at https://autbor.com/hellopy/**
- **Click the forward button to move through each step of the program's execution.**
- **You'll be able to see how the variables' values and the output change.**

## Your First Program

```
❶ # This program says hello and asks for my name.
❷ print('Hello, world!')
print('What is your name?') # ask for their name
❸ myName = input()
❹ print('It is good to meet you, ' + myName)
❺ print('The length of your name is:')
print(len(myName))
❻ print('What is your age?') # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

## Dissecting Your Program: Comments

❶ # This program says hello and asks for my name.

- Comment: the line following a hash mark (#)
- Any text for the rest of the line following a hash mark (#) is part of a comment
- Python ignores comments,

Python also ignores blank line after comment. Add any number of blank lines to program as you want. This can make your code easier to read, like paragraphs in a book.

- Why Comment?
  - ✓use them to write notes or
  - ✓remind yourself what the code is trying to do.
- Commenting out code: putting a '#' in front of a line of code to temporarily remove it while testing a program.
  - ✓ useful when trying to figure out why a program isn't working.
  - ✓ remove the # later when ready to put the line back in.

## The print() Function

➋ print('Hello, world!')

print('What is your name?') # ask for their name

- **The print() statement: is a function**
- **Purpose:** displays the string value inside its parentheses on the screen.
- **The line print('Hello, world!') means** “Print out the text in the string
- **When Python executes this line,**

✓it is calling the print() function and

✓the string value is being passed to the function.

## The print() Function

❷ print('Hello, world!')

print('What is your name?') # ask for their name

- **Argument:** A value that is passed to a function call.
- Notice that the quotes are not printed to the screen. They just mark where the string begins and ends; they are not part of the string value.
- You can also use this function to **put a blank line** on the screen; just call print() with nothing in between the parentheses.
- When you write a function name, the opening and closing parentheses at the end identify it as the name of a function.
- It is print() rather than print. (If not: error)

# The input() Function

❸ myName = input()

- **input() function** waits for the user to type some text on the keyboard and press ENTER.
- This function call
  - ✓ evaluates to a string equal to the user’s text(user’s text=string data type) and
  - ✓ The line of code assigns the myName variable to this string value.
- Consider input() function call as an expression that evaluates to whatever string the user typed in.
- If the user entered 'Al', then the expression would evaluate to myName = 'Al'.

# Printing the User’s Name

❹ print('It is good to meet you, ' + myName)

Expressions can always evaluate to a single value.

- If 'Al' is the value stored in myName on line ❸, then this expression evaluates to:
  - ✓ 'It is good to meet you, Al'.
- This single string value is then passed to print(), which prints it on the screen.

# The len() Function

❺ print('The length of your name is:')

print(len(myName))

- The len() function takes a string value (or a variable containing a string), and the function evaluates to the

  integer value equal to the number of characters in that

  string

Python does not have a character data type,
a single character =a string with a length of 1.

# The len() Function

```
❺ print('The length of your name is:')
   print(len(myName))
```

- The len() function takes a string value (or a variable containing a string), and the function evaluates to the integer value equal to the number of characters in that string

```
>>> len('hello')
5
>>> len('My Name')
7
>>> len('')
0
```

# The len() Function

```
❺ print('The length of your name is:')
   print(len(myName))
>>> len('My very energetic monster just scarfed nachos.')
46
```

- len(myName)
  - ✓evaluates to an integer and
  - ✓It is then passed to print() to be displayed on the screen.
- The print() function allows you to pass it either integer values or string values, but not this:

```
>>> print('I am ' + 29 + ' years old.')
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
print('I am ' + 29 + ' years old.')
TypeError: can only concatenate str (not "int") to str
```

- The print() function isn’t causing that error, but rather it’s the expression you tried to pass to print().
- You get the same error message if you type the expression into the interactive shell on its own.

```
❺ >>> 'I am ' + 29 + ' years old.'
Traceback (most recent call last):
File "<pyshell#7>", line 1, in <module>
'I am ' + 29 + ' years old.'
TypeError: can only concatenate str (not "int") to str
```

- Python gives an error because the + operator can only add two integers together or concatenate two strings.
- You can’t add an integer to a string.
- Fix this by using a string version of the integer instead.

## The str(), int(), and float() Functions

- To concatenate an integer with a string, convert integer into string
- Ex: >>> 'I am ' + 29 + ' years old.‘
- First get the value '29', which is the string form of 29.
- To get the string form of an integer value, pass it to the str() function. This will evaluate to a string value version of the integer.
- Ex:

```
>>> str(29)      #converting integer into string(=Type casting)
'29‘             # string form of 29
```

## The str(), int(), and float() Functions

>>> 'I am ' + str(29) + ' years old.‘

I am 29 years old.      #O/P    ???????????????

- str(29) evaluates to '29',
- The expression 'I am ' + str(29) + 'years old.' evaluates to

 'I am ' + '29' + ' years old.', which in turn evaluates to 'I am 29 years old.'.

- >>> print('I am ' + 29 + ' years old.‘)
- The above evaluated value is then passed to the print() function.

# The str(), int(), and float() Functions

str(): Evaluates the value you pass to the string data type

int(): Evaluates the value you pass to the int data type

float(): Evaluates the value you pass to the floating-point data type.

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
```

## The str(), int(), and float() Functions

str(): Evaluates the value you pass to the string data type

int(): Evaluates the value you pass to the int data type

float(): Evaluates the value you pass to the floating-point data type.

```
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

## The str(), int(), and float() Functions

- Call the str() function and pass other data types: to obtain a string
- Call the int() function and pass other data types: to obtain a integer form of passed value.
- Call the float() function and pass other data types: to obtain a floating-point form of passed value.
- The str() function is handy when an integer or float value has to be concatenated to a string.
- The int() function is helpful when a number as a string value to be used in mathematics.
- **Ex: input() function always returns a string, even if user enters a number.**

# The str(), int(), and float() Functions

>>> spam = input() and enter 101 when it waits for your text.

>>> print(spam)

'101'.      #The value stored inside spam isn’t integer 101 but the string

- If you want to do mathematic calculation using the value in spam, use the int() function to get the integer form of spam and then store this as the new value in spam.

```
>>> spam = int(spam)
>>> spam
101
```

- Now spam variable has an integer value instead of a string.

## The str(), int(), and float() Functions

```
>>> spam * 10 / 5
202.0
```

• If you pass a value to int() function that Python cannot evaluate as an integer, Python will display an error message.

```
>>> int('99.99')     # first convert into float else throws an error
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int(float('99.99'))
99
```

# The str(), int(), and float() Functions

```
>>> int('twelve')
Traceback (most recent call last):
File "<pyshell#19>", line 1, in <module>
int('twelve')
ValueError: invalid literal for int() with base 10: 'twelve‘
```

- The int() function is also used to round a floating-point number down.

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

# The str(), int(), and float() Functions

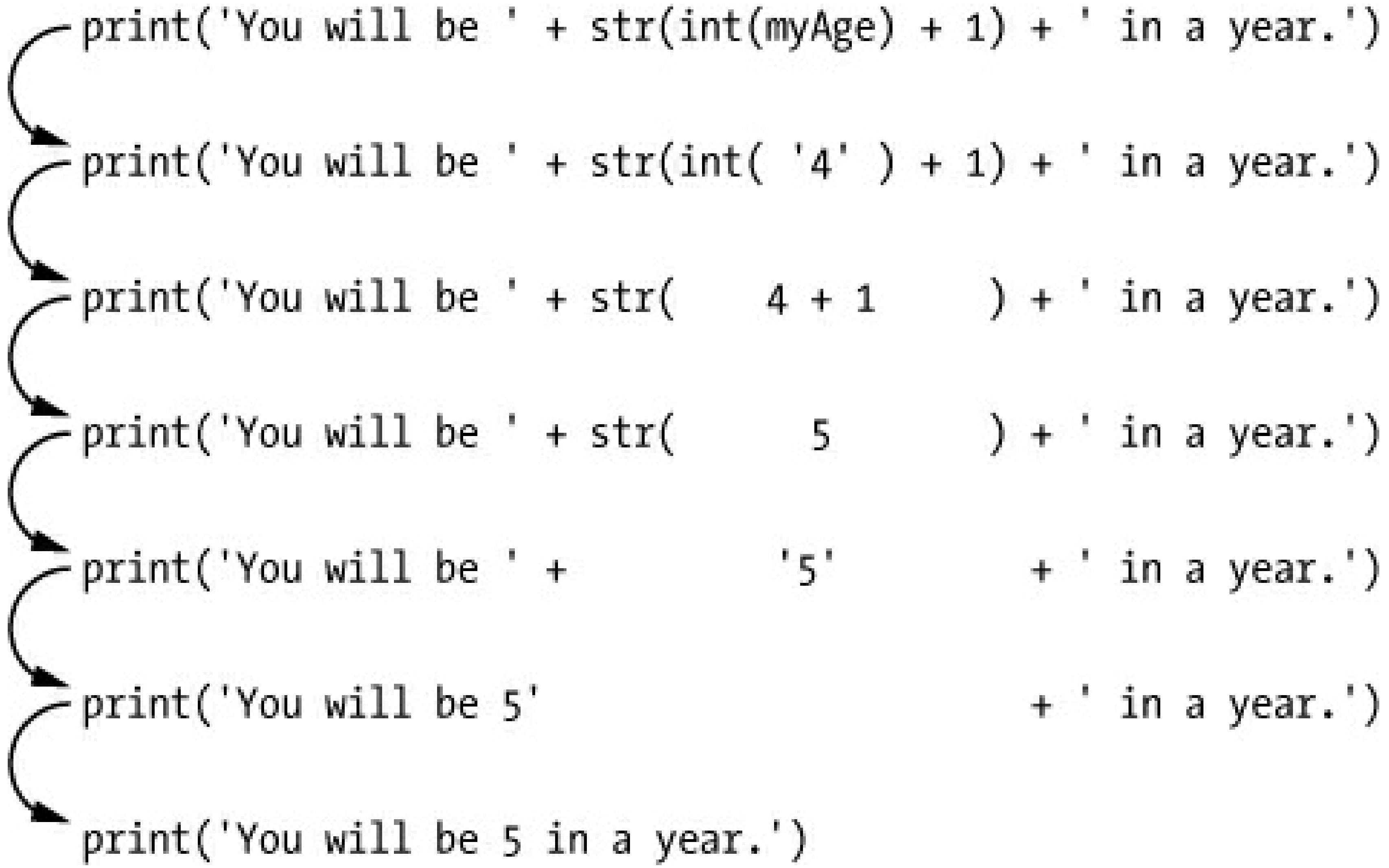
```
print('What is your age?')                    # ask for their age
myAge = input()                               #myAge has string value.
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

- An integer can be equal to a floating point form of that number.
- String value of a number is not = that number

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

# The str(), int(), and float() Functions

- If user enters the string '4' for myAge



```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')

print('You will be ' + str(int( '4' ) + 1) + ' in a year.')

print('You will be ' + str(    4 + 1     ) + ' in a year.')

print('You will be ' + str(      5       ) + ' in a year.')

print('You will be ' +          '5'        + ' in a year.')

print('You will be 5'                      + ' in a year.')

print('You will be 5 in a year.')
```

# FLOW CONTROL

- Program is just a series of instructions.
- But programming's real strength isn't simply executing every line, straight to the end from beginning, just running one instruction after another.
- Based on how expressions evaluate, a program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you
- Flow control statements can decide which Python instructions to execute under which conditions.

## Boolean Values

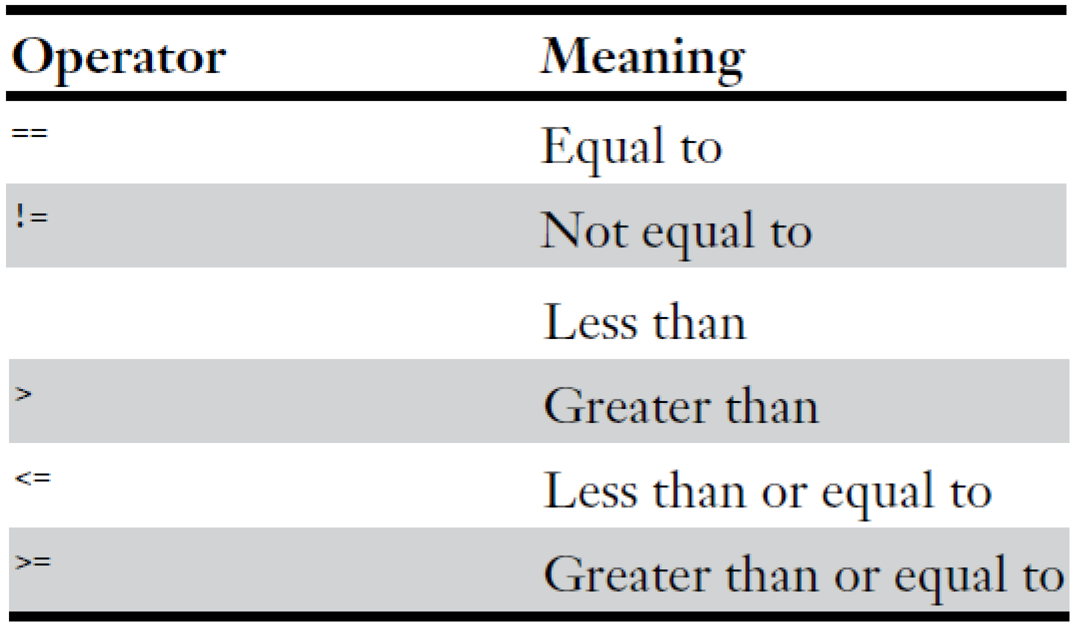
- Integer, floating-point, and string data types have an unlimited number of possible values,
- Boolean data type has only two values: True and False.
- Boolean data type is named after mathematician George Boole
- Don’t use the quotes  for Boolean values True & False in Python code.
- They always start with a capital T or F, with rest in lowercase.

## Boolean Values

```
➊ >>> spam = True
>>> spam
True
➋ >>> true
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module> true
NameError: name 'true' is not defined
➌ >>> True = 2 + 2
SyntaxError: can't assign to keyword
```

# Comparison Operators

- Comparison operators compare two values and evaluate down to a single Boolean value
- They are also called as relational operators



| Operator | Meaning |
|---|---|
| == | Equal to |
| != | Not equal to |
|  | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |

- These operators evaluate to True or False depending on the values entered

# Comparison Operators

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

- == (equal to equal to) evaluates to True when the values on

both sides are the same,

- != (not equal to) evaluates to True when the two values are different.
- The == and != operators can actually work with values of any data type.

# Comparison Operators

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
❶ >>> 42 == '42'
False.
```

- because Python considers the integer 42 to be different from string '42'.

## Comparison Operators

- The <, >, <=, and >= operators, work properly only with integer and floating-point values.

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
❶ >>> eggCount <= 42
True
>>> myAge = 29
❷ >>> myAge >= 10
True
```

## Comparison Operators

- THE DIFFERENCE BETWEEN THE == AND = OPERATORS:

| == OPERATOR | = OPERATOR |
|---|---|
| This is a comparison operator | This is an assignment operator |
| This has two equal signs | This has just one equal sign |
| asks whether two values are the same as each other | puts the value on the right into the Variable on the left. |

# Boolean Operators

- The three Boolean operators (and, or, and not) are used to compare
Boolean values.
- Like comparison operators, they evaluate these
expressions down to a Boolean value.

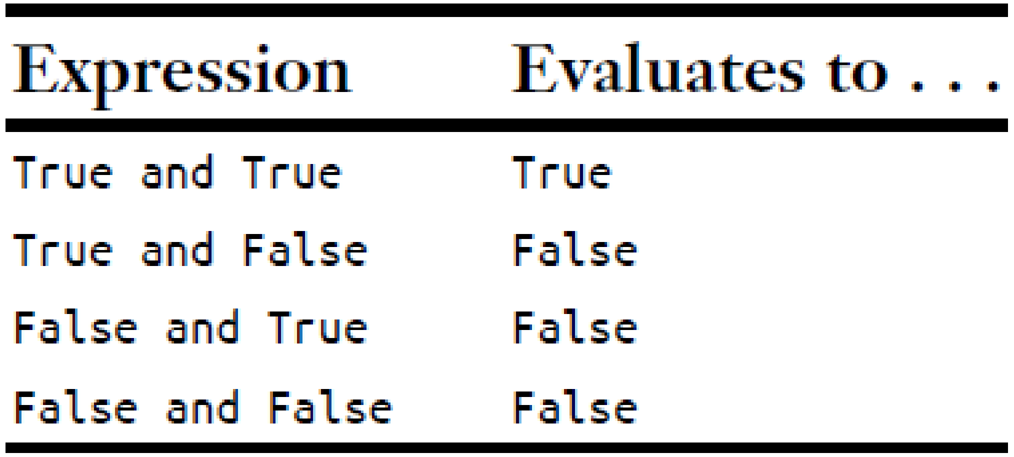
- **Binary Boolean Operators:** and,     or operators
- They always take two Boolean values (or expressions).
- The and operator evaluates an expression to True if both Boolean values are True; otherwise, it evaluates to False.

```
>>> True and True
True
>>> True and False
False
```

# Boolean Operators

```
>>> True and True
True
>>> True and False
False
```

- and Operator’s Truth Table



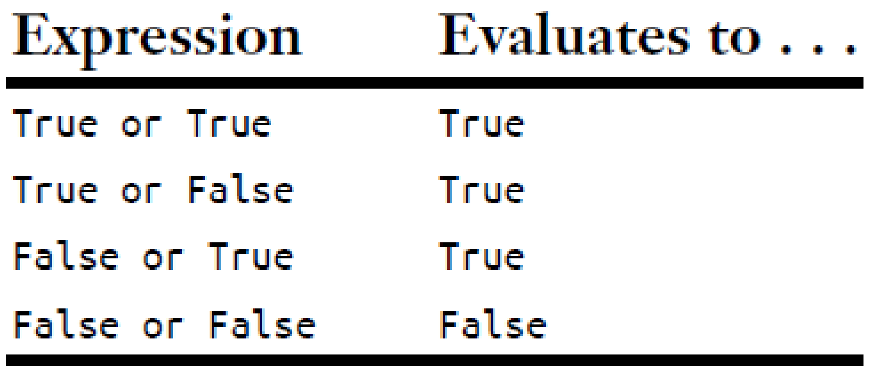
| Expression | Evaluates to . . . |
|---|---|
| `True and True` | `True` |
| `True and False` | `False` |
| `False and True` | `False` |
| `False and False` | `False` |

## Boolean Operators

- On the other hand, the or operator evaluates an expression to True if either of the two Boolean values is True.

```
>>> False or True
True
>>> False or False
False                    #If both are False, it evaluates to False.
```

- or Operator’s Truth Table



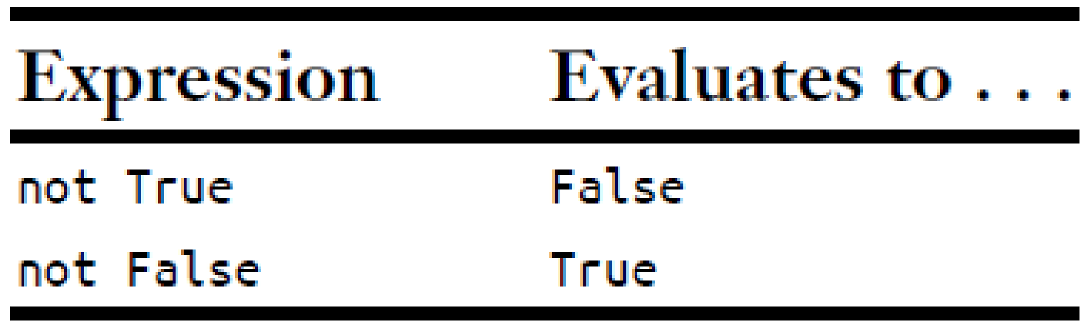
| Expression | Evaluates to . . . |
|---|---|
| `True or True` | `True` |
| `True or False` | `True` |
| `False or True` | `True` |
| `False or False` | `False` |

## Boolean Operators

- The not Operator:
  - ✓ not operator operates on only one Boolean value (or expression).
  - ✓ This makes it a unary operator.
  - ✓ The not operator simply evaluates to the opposite Boolean value.

```
>>> not True
False
❶ >>> not not not not True    #nest not operators ❶
True
```

not Operator's Truth Table:



| Expression | Evaluates to . . . |
|---|---|
| `not True` | `False` |
| `not False` | `True` |

## Mixing Boolean and Comparison Operators

- Since the comparison operators evaluate to Boolean values, use them in expressions with the Boolean operators.
- and, or, and not operators are called Boolean operators because they always operate on the Boolean values True and False.
- While expressions like 4 < 5 aren’t Boolean values, they are expressions that evaluate down to Boolean values.

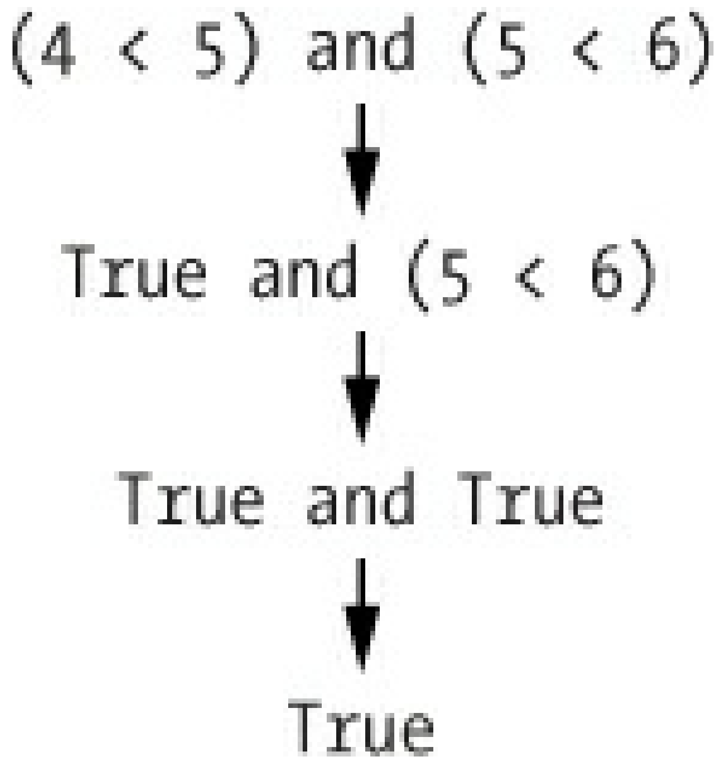
Ex:

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

## Mixing Boolean and Comparison Operators

- The computer will evaluate the left expression first, and then it will evaluate the right expression.
- When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value.

Ex:



```
(4 < 5) and (5 < 6)
True and (5 < 6)
True and True
True
```

## Mixing Boolean and Comparison Operators

- Multiple Boolean operators can be used in an expression, along with the comparison operators

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

- order of operations for the Boolean operators:

1. Python evaluates any math and comparison operators first,
2. Evaluates the not operators,
3. then the and operators,
4. then the or operators.

# Elements of Flow Control

**Condition:** Starting part of Flow Control Statements
**Clause:** a block of code following the Condition.

Conditions:

- They are the Boolean expressions, in the context of flow control statements
- Conditions always evaluate down to a Boolean value, True or False

- Every flow control statement uses a condition and

Flow control statement decides what to do based on whether
its condition is True or False.

## Blocks of Code

```
name = 'Mary'
password = 'swordfish'
if name == 'Mary':
    ❶ print('Hello, Mary')
    if password == 'swordfish':
        ❷ print('Access granted.')
    else:
        ❸ print('Wrong password.')
```

- The first block of code ❶ starts at the line print('Hello, Mary') and contains all the lines after it.
- Inside this block is another block ❷, which has only a single line in it: print('Access Granted.').
- The third block ❸ is also one line long: print('Wrong password.').
- View the execution of this program at https://autbor.com/blocks/.

## Program Execution

- Python starts executing instructions at the top of the program going down, one after another.
- The program execution is a term for the current instruction being executed.
- If you print the source code on paper and put your finger on each line as it is executed, you can think of your finger as the program execution.
- Program with flow control statements: If you use your finger to trace this, you’ll find yourself jumping around the source code based on conditions, and you’ll probably skip entire clauses.

# Flow Control Statements

- They are the actual decisions which programs will make.

## if Statements

- If statement’s condition is True: An if statement’s clause (=block following the if statement) will execute
- If statement’s condition is False : The clause is skipped

- In plain English, if statement could be read as,

“If this condition is true, execute the code in the clause.”

# Flow Control Statements

if statement consists of the following:

- The if keyword
- A condition (an expression that evaluates to True or False)
- A colon
- Starting on the next line, an indented block of code (= if clause)

Ex: Consider a code that checks to see whether someone’s name is Alice.

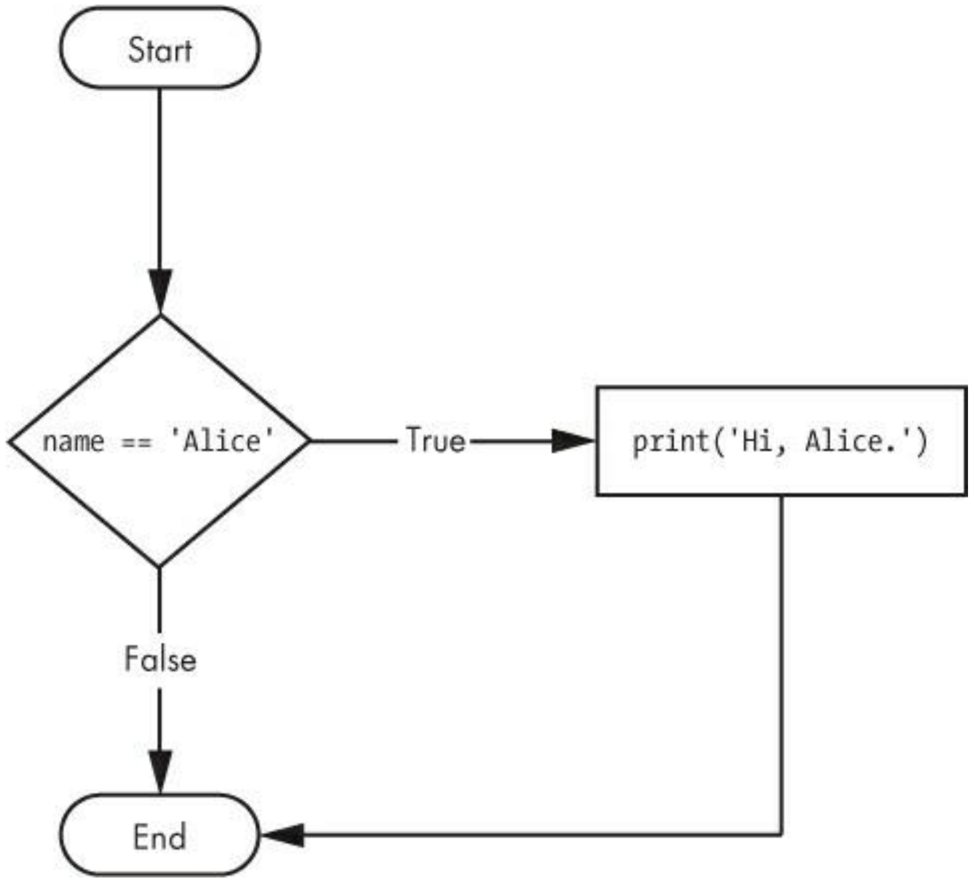
```
name=input(‘Enter your name:  ‘)
If name==‘Alice’:
        print(‘Hi! Alice’)
```

# Flow Control Statements

```
name=input(‘Enter your name:  ‘)
if name==‘Alice’:
        print(‘Hi! Alice’)
```

- All flow control statements end with a colon and
- They are followed by a new block of code (the clause).

This if statement’s clause:  block with print('Hi, Alice.').



## else Statements

- else statement optionally follows if clause
- Else clause is executed only when the if statement's condition is False.
- In plain English, else statement could be read as, "If this condition is true, execute this code.
  Or else, execute that code."
- An else statement doesn't have a condition

## else Statements

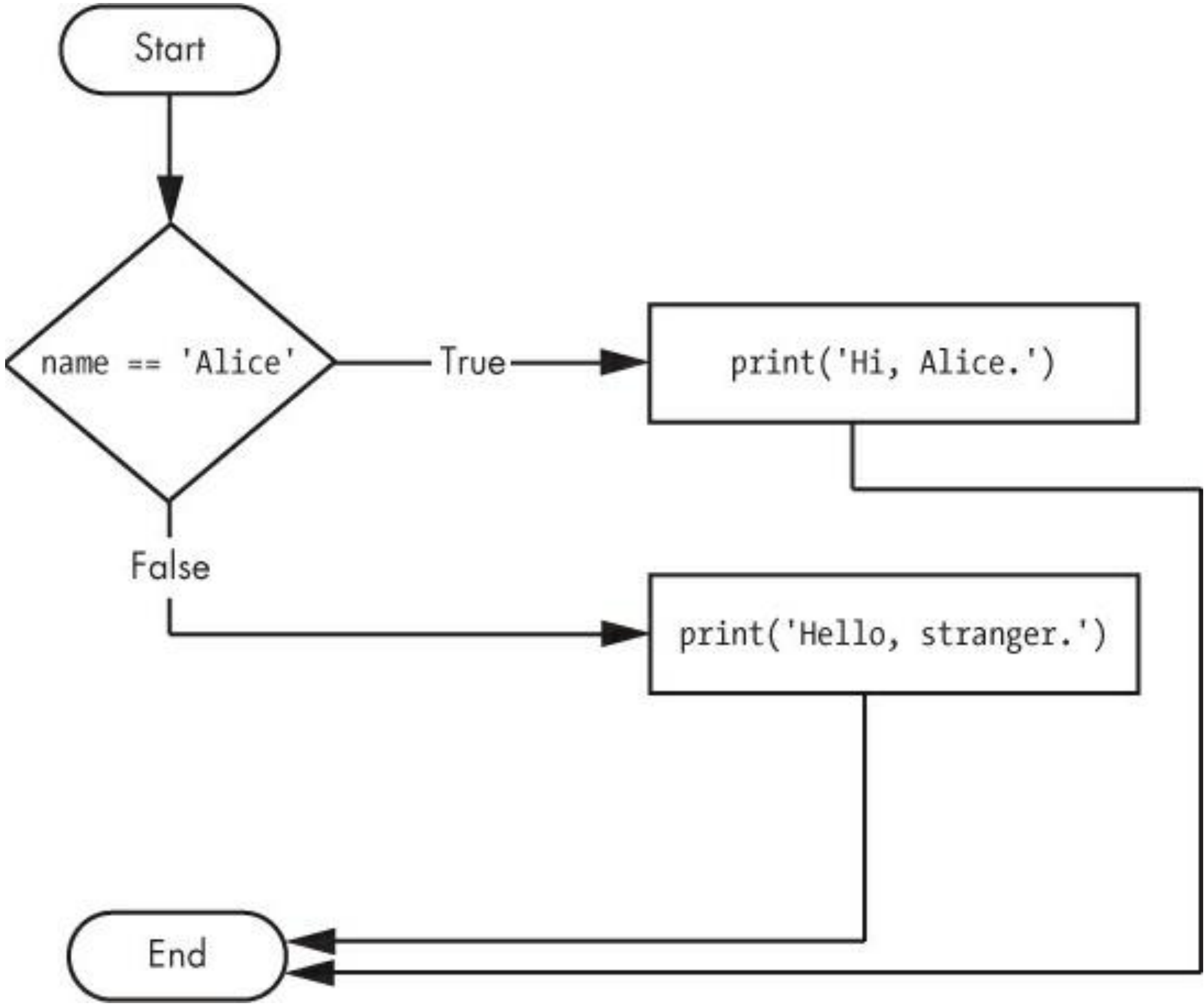
- An else statement doesn't have a condition

- else statement consists of the following:
  - ✓The else keyword
  - ✓A colon
  - ✓Starting on the next line, an indented block of code (=else clause)

```
if name == 'Alice':
        print('Hi, Alice.')
else:
        print('Hello, stranger.')
```

## else Statements

```
if name == 'Alice':
        print('Hi, Alice.')
else:
        print('Hello, stranger.')
```



# Elif Statements

- elif statement: use when you want one of many possible clauses to execute.
- It is an "else if" statement.
- It always follows an if or another elif statement.
- It provides another condition that to be checked only if all of the previous conditions were False.
- elif statement always consists of the following:
  - ✓The elif keyword
  - ✓A condition (=an expression that evaluates to True or False)
  - ✓A colon
  - ✓Starting on the next line, an indented block of code (= elif clause)

# elif Statements

```
if name == 'Alice':
        print('Hi, Alice.')
elif age < 12:
        print('You are not Alice, kiddo.')
```

- Two conditions are checked name and person’s age, and the program will tell something different if they’re younger than 12.
- elif clause executes, if age < 12 is True & name == 'Alice' is False.
- However, if both of the conditions are False, then both of the clauses are skipped.
- It is not guaranteed that at least one of the clauses will be

executed. When there is a chain of elif statements, only one or none of the clauses will be executed.

- Once one of the statements’ conditions is found to be True, the rest of the elif clauses are automatically skipped.

## elif Statements

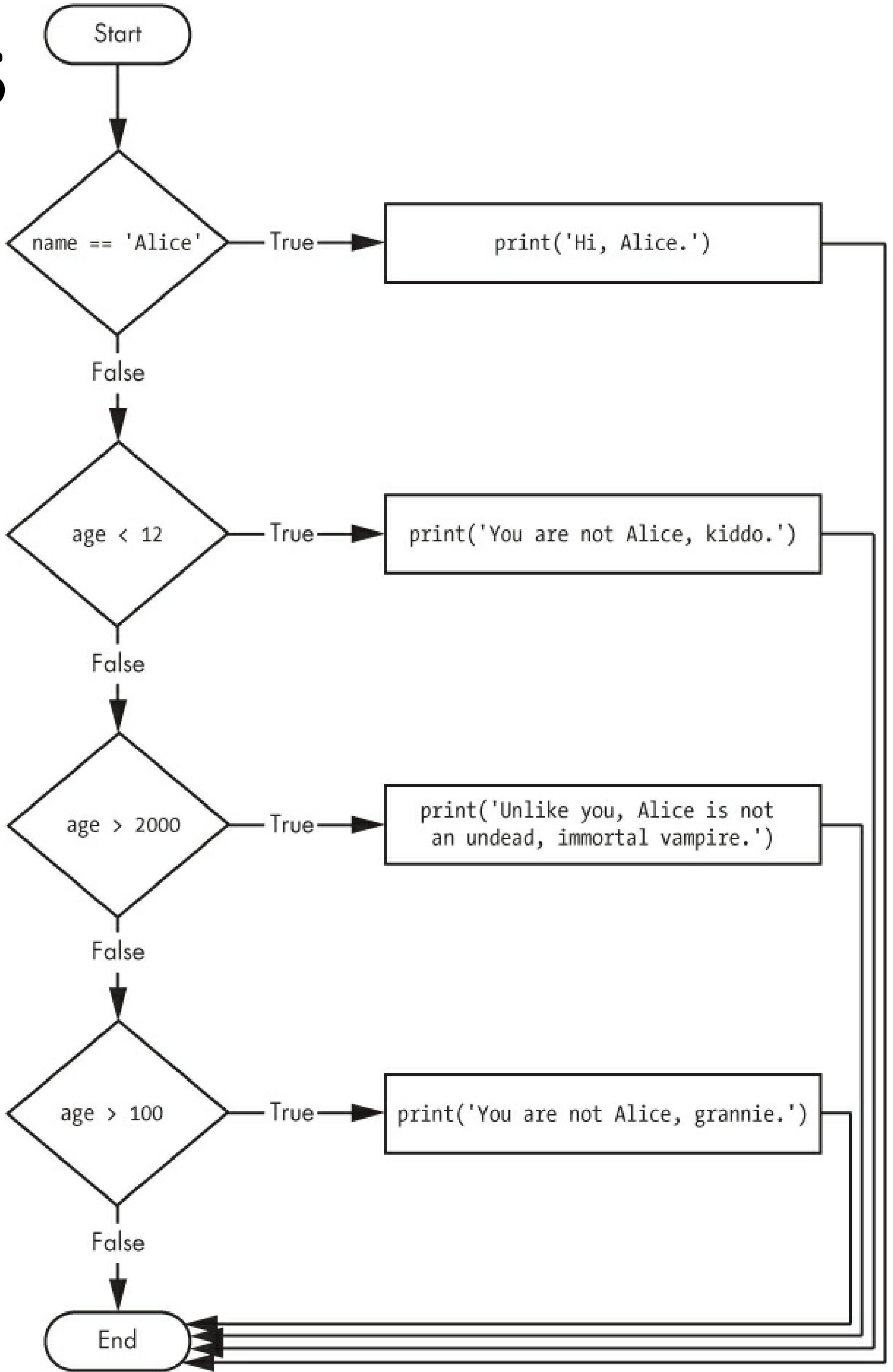
- Ex: consider a program with name vampire.py

```
name = 'Carol'
age = 3000
if name == 'Alice':
        print('Hi, Alice.')
elif age < 12:
        print('You are not Alice, kiddo.')
elif age > 2000:
        print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
        print('You are not Alice, grannie.')
```

View the execution of this program at https://autbor.com/vampire/.

# elif Statements

```
name = 'Carol'
age = 3000
if name == 'Alice':
        print('Hi, Alice.')
elif age < 12:
        print('You are not Alice, kiddo.')
elif age > 2000:
        print('Unlike you, Alice is not an undead,
              immortal vampire.')
elif age > 100:
        print('You are not Alice, grannie.')
```



Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
age > 2000
True
print('Unlike you, Alice is not an undead, immortal vampire.')
False
age > 100
True
print('You are not Alice, grannie.')
False
End

# elif Statements

- Order of the elif statements matters?
- Rearrange statements in in vampire.py to introduce a bug.
- Imp: rest of the elif clauses are automatically skipped once a True condition ha been found.

```
name = 'Carol'
age = 3000
if name == 'Alice':
        print('Hi, Alice.')
elif age < 12:
        print('You are not Alice, kiddo.')
❶ elif age > 100:
        print('You are not Alice, grannie.')
elif age > 2000:
        print('Unlike you, Alice is not an undead, immortal vampire.')
```

age > 100 condition is True (after all, 3,000 is greater than 100) ❶, the string 'You are not Alice, grannie.' is printed

Expected o/P: 'Unlike you, Alice is not an undead, immortal vampire.'.

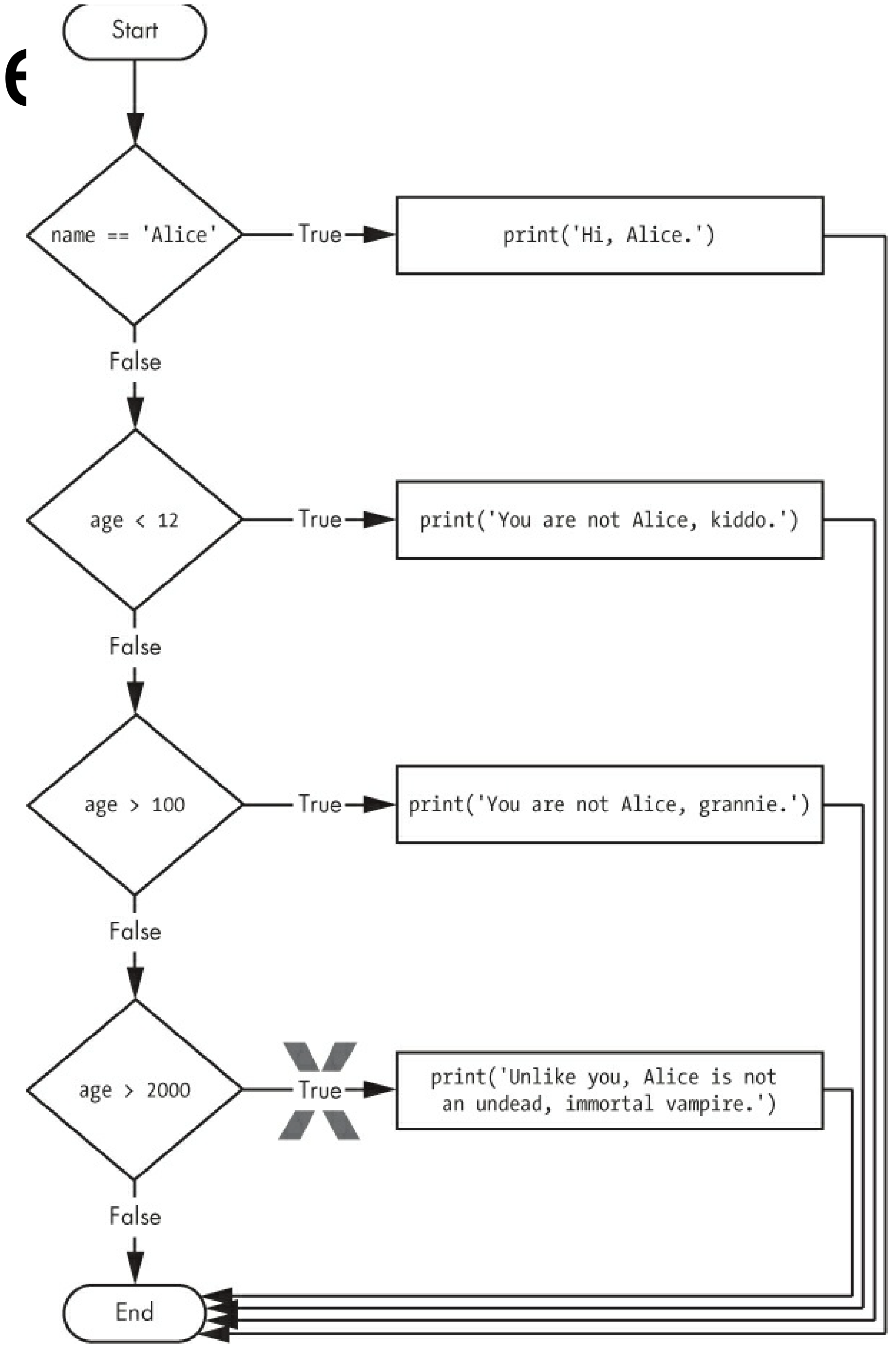
Actual o/P: 'You are not Alice, grannie.'

view the execution of this program at https://autbor.com/vampire2/.

# Elif and else Stateme

Note: how the diamonds for
age > 100 and age > 2000 are swapped.

To ensure that at least one (and only one) of the clauses will be executed :
Have an else statement after the last elif statement.
In that case, it is guaranteed that: Suppose conditions in every if and elif statement are False,
then the else clause is executed.


Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
age > 100
True
print('You are not Alice, grannie.')
False
age > 2000
True
print('Unlike you, Alice is not an undead, immortal vampire.')
False
End

# Elif and else Statements

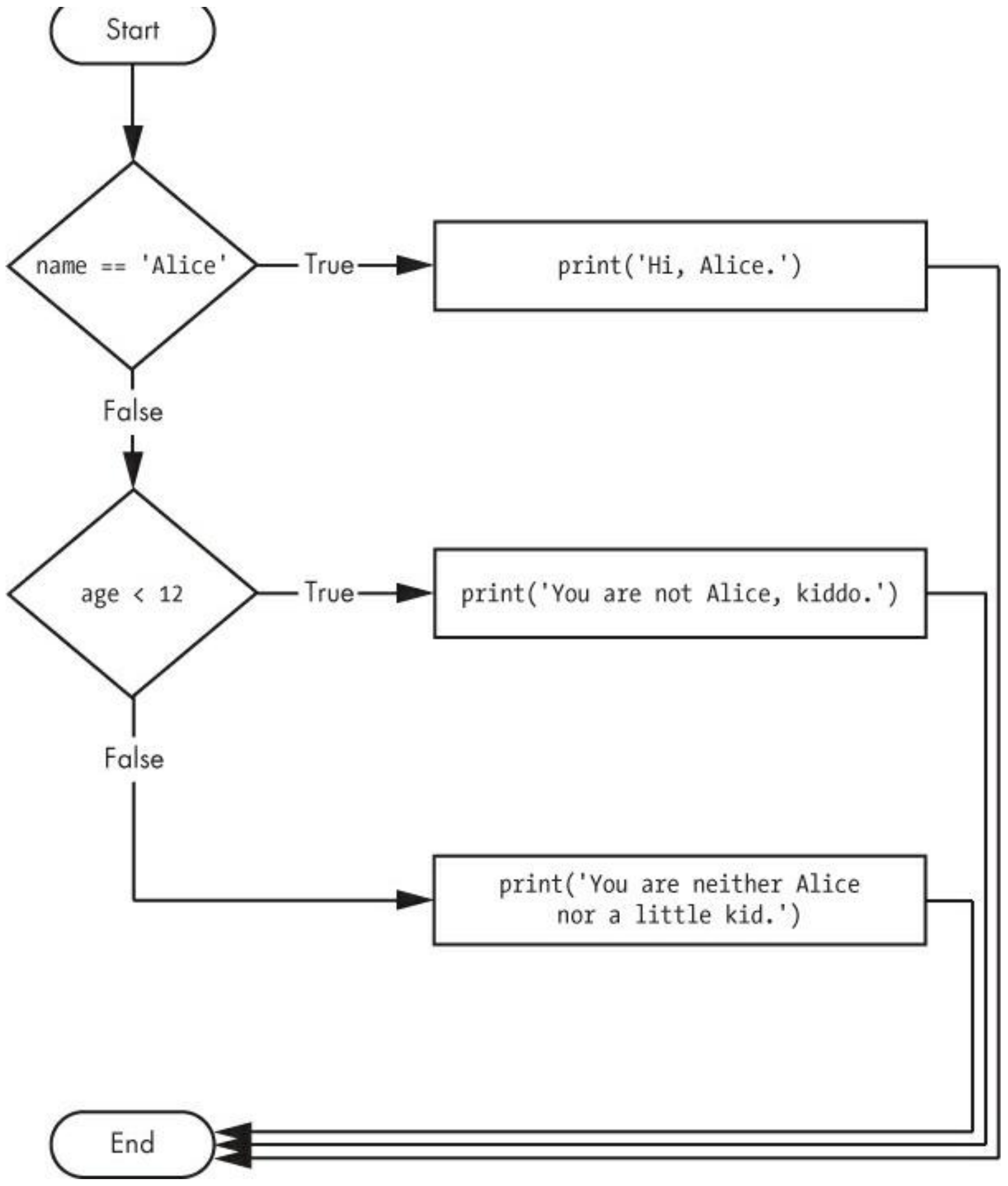
Ex: the Alice program to use if, elif, and else clauses:

```
name = 'Carol'
age = 3000
if name == 'Alice':
        print('Hi, Alice.')
elif age < 12:
        print('You are not Alice, kiddo.')
else:
        print('You are neither Alice nor a little kid.')
```

View the execution of this program at https://autbor.com/littlekid/.

# else Statements

- flowchart for this new code, littleKid.py.
- In plain English:
- “If the first condition is true, do this.
- Else, if the second condition is true, do that. Otherwise, do something else.”
- Imp: When you use if, elif, and else
- statements together, remember these rules about how to order them to avoid bugs like the one in vampire.py.



Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
print('You are neither Alice nor a little kid.')
End

# Elif and else Statements

Summary:

- First, there is always exactly one if statement.
- Any elif statements should follow the if statement.
- Second, if you want to be sure that at least one clause is executed, close the structure with an else statement.

# While Loop Statements

Use of While Statement: To execute a block of code over and over again.

- ➢ The code in a while clause will be executed as long as the while statement's condition is True.
- ➢ Code in a while statement consists of :
  - ✓The while keyword
  - ✓A condition (=an expression that evaluates to True or False)
  - ✓A colon
  - ✓Starting on th next line, an indented block of code (called while clause)

## While Loop Statements

- a while statement looks similar to an if statement.
- The difference is in how they behave.
- while clause is often called the while loop or just the loop.

Ex: if statement:

At the end of an if clause, the program execution continues after the if statement.

```
spam = 0
if spam < 5:
        print('Hello, world.')
spam = spam + 1
```

O/P : Hello, world.

Ex: while statement:

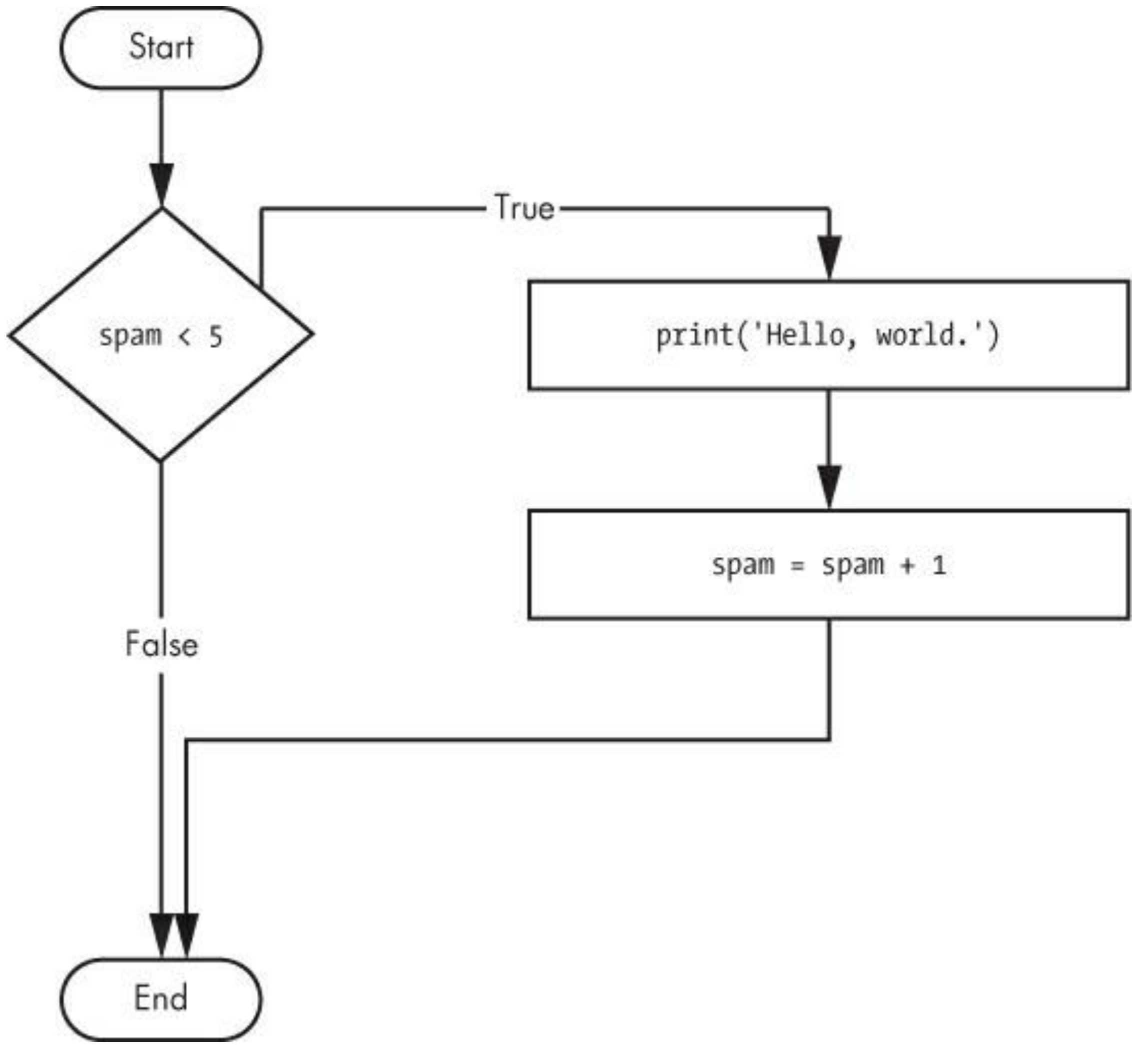
at the end of a while clause, the program execution jumps back to the start of the while statement

```
spam = 0
while spam < 5:
        print('Hello, world.')
spam = spam + 1
```

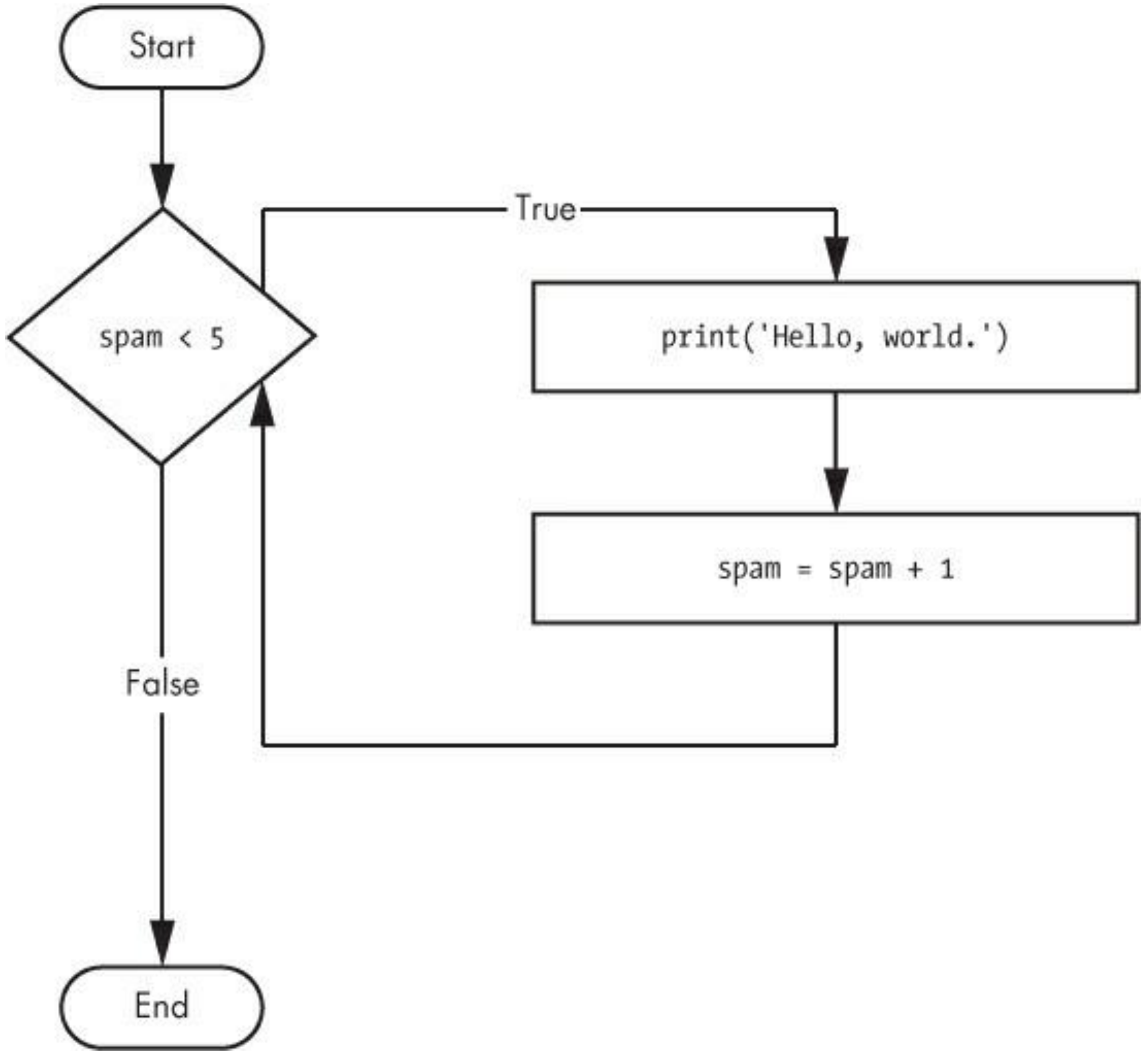
O/P : Hello, world repeated five times!

## While Loop Statements

Flowchart for the if statement code



flowchart for while statement code



# While Loop Statements

if statement checks the condition, and it prints
Hello, world. only once if that condition is true. The code with the while loop, on the other hand, will print it five times. The loop stops after five prints because the integer in spam increases by one at the end of each loop iteration, which means that the loop will execute five times before spam < 5 is False.

In the while loop, the condition is always checked at the start of each iteration (that is, each time the loop is executed). If the condition is True, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be False, the while clause is skipped.

# While Loop Statements

- if statement checks the condition, and it prints Hello, world, only once if that condition is true.

- The code with the while loop, will print it five times.
- The loop stops after five prints because the integer in spam increases by one at the end of each loop iteration, which means that the loop will execute five times before spam < 5 is False.

- In the while loop, the condition is always checked at the start of each iteration (that is, each time the loop is executed).
- If the condition is True, then the clause is executed, and afterward, the condition is checked again.
- The first time the condition is found to be False, the while clause is skipped.

# Annoying while Loop

An Annoying while Loop:

Ex: program that will keep asking to type, your name.

```
❶ name = ''
❷ while name != 'your name':
          print('Please type your name.')
❸         name = input()
❹ print('Thank you!')
```

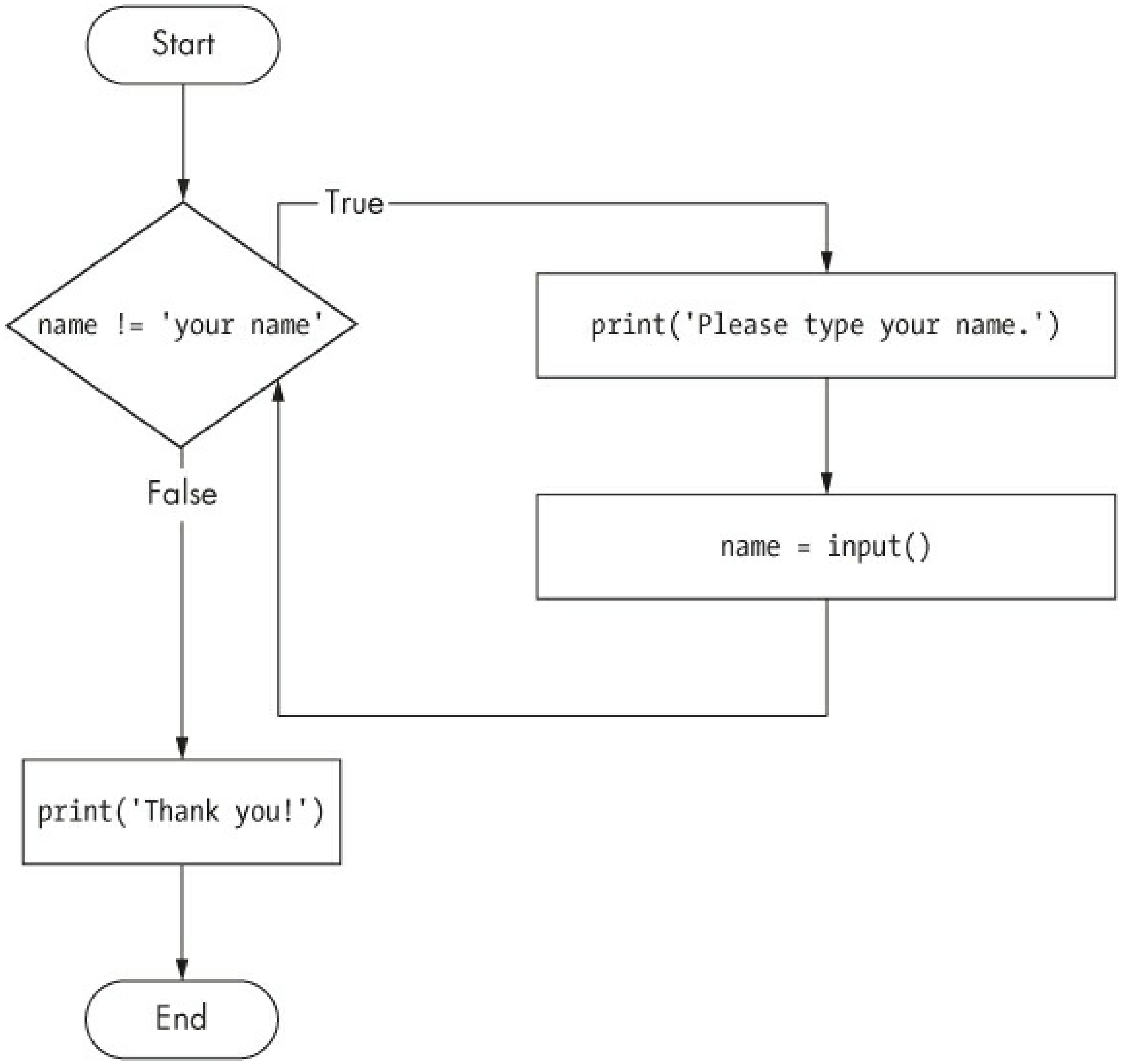
view the execution of this program at https://autbor.com/yourname/.

# Annoying while Loop

- ➢ First, the program sets the name variable ❶ to an empty string.
- ➢ This is so that the name != 'your name' condition will evaluate to True and the program execution will enter the while loop’s clause ❷.
- ➢ The code inside this clause asks the user to type their name, which is assigned to the name variable ❸. Since this is the last line of the block, the execution moves back to the start of the while loop and reevaluates the condition. If the value in name is not equal to the string 'your name', then the condition is True, & execution enters the while clause again.
- ➢ But once the user types your name, the condition of the while loop will be 'you name' != 'your name', which evaluates to False. The condition is now False, & instead of program execution reentering the while loop’s clause, Python skips past it & continues running rest of program ❹.

# Annoying while Loop

```
Please type your name.
Al
Please type your name.
Albert
Please type your name.
%#@#%*(^&!!!
Please type your name.
your name
Thank you!
```



## Annoying while Loop

If you never enter your name????

Then the while loop’s condition will never
be False, and the program will just keep asking forever.

Here, the input() call lets the user enter the right string to make the program move on.

How to break out of a while loop???

# break Statements

- ➢ Break Statement: a shortcut to break out of a while loop's clause early.
- ➢ If the execution reaches a break statement, it immediately exits the while loop's clause.
- ➢ break statement: use keyword in code: break
- ➢ Consider a program: Name2.py
- ➢ uses a break statement to escape the loop:

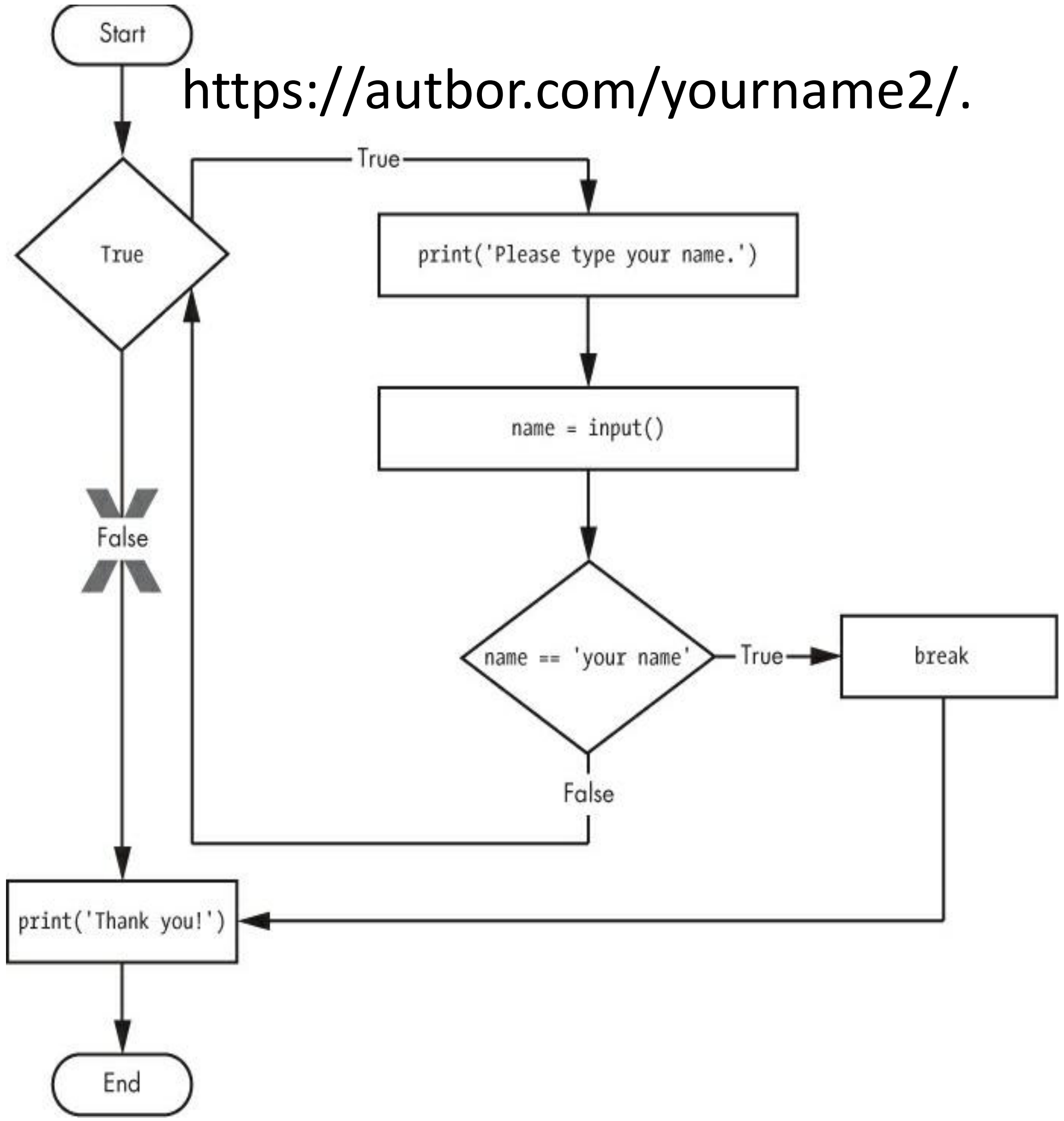
```
❶ while True:
        print('Please type your name.')
❷       name = input()
❸       if name == 'your name':
❹               break
❺ print('Thank you!')
```

# break Statements

➢ Name2.py

```
❶ while True:    #program starts
        print('Please type your name.')
❷       name = input()
❸       if name == 'your name':
❹               break
❺ print('Thank you!')      #ends
```

https://autbor.com/yourname2/.



# break Statements

➢ Name2.py

```
❶ while True:    #program starts
        print('Please type your name.')
❷       name = input()
❸       if name == 'your name':
❹               break
❺ print('Thank you!')      #ends
```

The first line ❶ creates an infinite while loop
Because condition is always True. (expression True, always evaluates down to value True.) (An infinite loop that never exits is a common programming bug.)

- ➢ Program asks user to enter your name ❷.
- ➢ Execution is still inside the while loop, an if statement checks ❸ whether name is equal to 'your name'.
- ➢ If this condition is True, the break statement is run ❹, and the execution moves out of the loop to print('Thank you!') ❺.

# break Statements

The first line ❶ creates an infinite while loop Because condition is always True.

- Program asks user to enter your name ❷.
- Execution is still inside the while loop, if statement checks ❸ whether name is equal to 'your name'.
- If this condition is True, the break statement is run ❹, and the execution moves out of the loop to print('Thank you!') ❺.

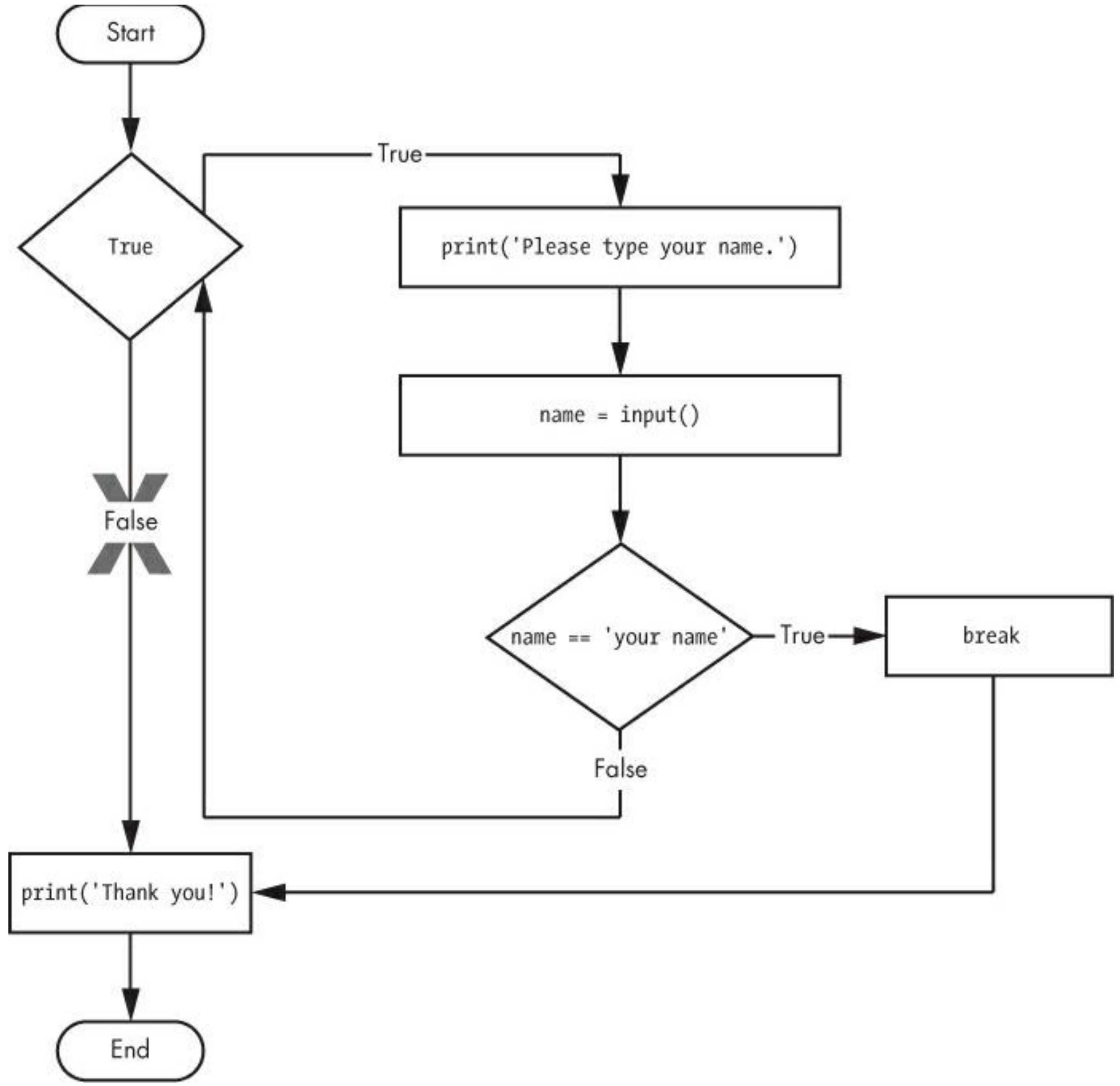
- Name2.py

```
❶ while True:    #program starts
        print('Please type your name.')
❷       name = input()
❸       if name == 'your name':
❹               break
❺ print('Thank you!')     #ends
```

- Otherwise, the if statement's clause that contains the break statement is skipped, which puts the execution at the end of the while loop.
- At this point, the program execution jumps back to the start of the while statement ❶ to recheck the condition.
- Since this condition is merely True Boolean value, the execution enters the loop to ask the user to type your name again

## break Statements

- Run yourName2.py, and enter the same text you entered for
  Your Name.py
- The rewritten program should respond in the same way as the original.
- Note that the
  X path will logically never happen, because the loop condition is always True.



## TRAPPED IN AN INFINITE LOOP?

Create a program:      infiniteLoop.py

```
while True:
    print('Hello, world!')
```

Press CTRL-C or
  select Shell ‣ Restart Shell from IDLE’s menu

This will send a KeyboardInterrupt error to your program and cause it to
  stop immediately.

When run this program, it will print Hello, world! to screen
  forever because while statement’s condition is always True.

CTRL-C is also handy if you want to simply terminate your
  program immediately, even if it’s not stuck in infinite loop.

# continue Statements

- Continue statements are used inside loops
- When the program execution reaches a continue statement, the program execution
- Immediately jumps back to the start of the loop and reevaluates the loop’s condition.
- (This is also what happens when the execution reaches the end of the loop.)

- Consider a program that asks for a name and Password, as swordfish.py

# swordfish.py

```
while True:
        print('Who are you?')
        name = input()
        ❶ if name != 'Joe':
                ❷ continue
        print('Hello, Joe. What is the password? (It is a fish.)')
        ❸ password = input()
                if password == 'swordfish':
                        ❹ break
❺ print('Access granted.')
```

- Run this program:
- Until you claim to be Joe, the program shouldn’t ask for a pwd
- once you enter correct password, it should exit.

# break Statements

- If the user enters any name besides Joe ❶, the continue statement ❷ causes the program execution to jump back to the start of the loop.
- When the program reevaluates the condition, the execution will always enter the loop, since condition is = value True.
- Once the user makes it past that if statement, they are asked for a password ❸.
- If the password entered is swordfish, then the break statement ❹ is run, and the execution jumps out of the while loop to print Access granted ❺.
- Otherwise, the execution continues to the end of the while loop, where it then jumps back to the start of the loop.

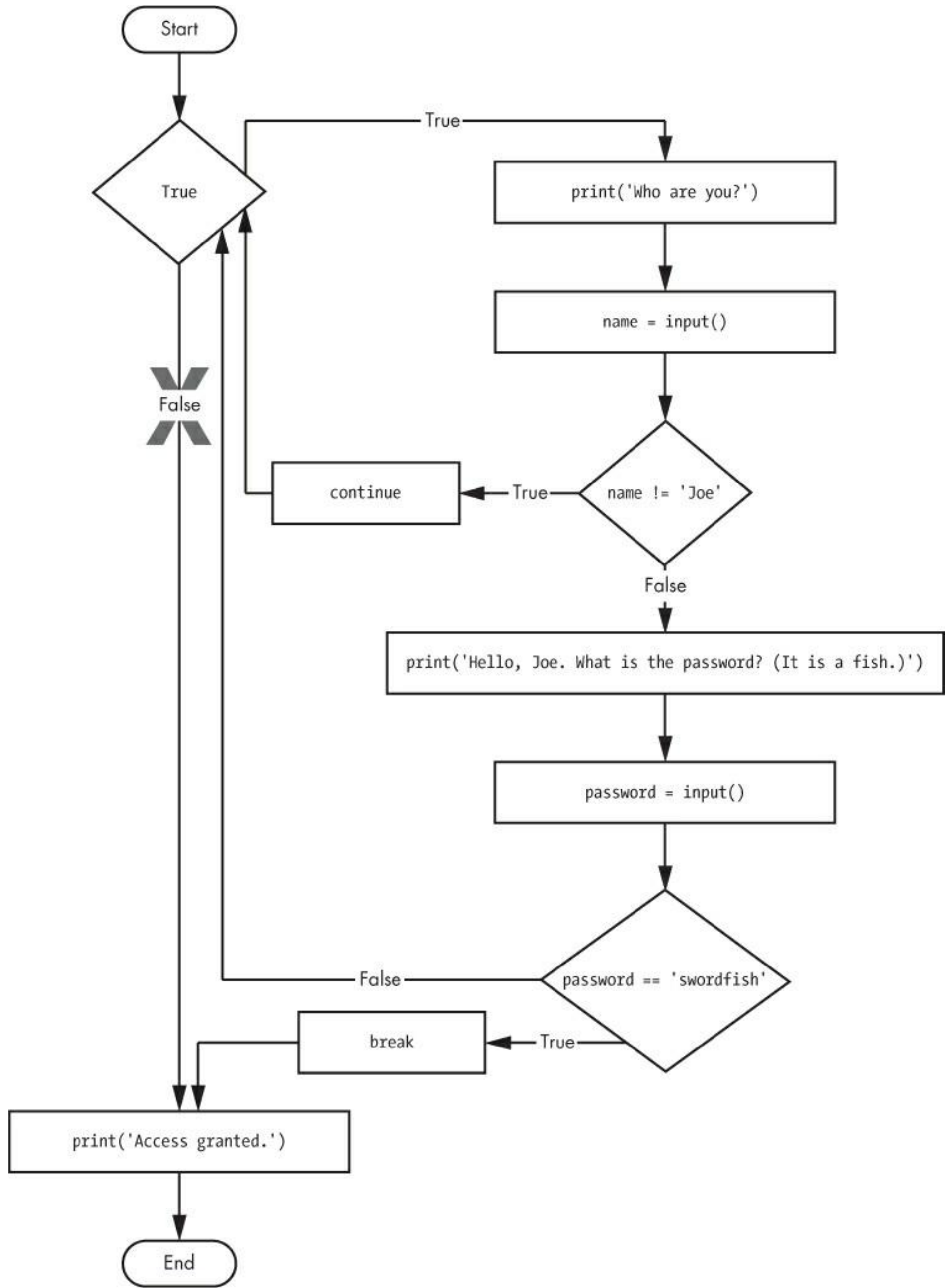
```
while True:
        print('Who are you?')
        name = input()
        ❶ if name != 'Joe':
                ❷ continue
        print('Hello, Joe. What is the pwd?
        ❸ password = input()
                if password == 'swordfish':
                        ❹ break
❺ print('Access granted.')
```

- ➢ Run this program:
- ➢ Until you claim to be Joe,
the program shouldn’t ask for a pwd
- ➢ once you enter correct
password, it should exit.

## break Statements

swordfish.py

- If the user enters any name besides Joe ❶, the continue statement ❷ causes the program execution to jump back to the start of the loop.
- When the program reevaluates the condition, the execution will always enter the loop, since condition is = value True.
- Once the user makes it past that if statement, they are asked for a password ❸.
- If the password entered is swordfish, then the break statement ❹ is run, and the execution jumps out of the while loop to print Access granted ❺.
- Otherwise, the execution continues to the end of the while loop, where it then jumps back to the start of the loop.



## “TRUTHY” AND “FALSEY” VALUES

- Conditions will consider some values equivalent to True and False.
- When used in conditions,
  - ✓0, 0.0, and ‘’ (the empty string) are considered False,
  - ✓all other values are considered True.
- Ex, consider program howmanyguests:

```
Name=‘’
❶ while not name:
        print('Enter your name:')
        name = input()
   print('How many guests will you have?')
   numOfGuests = int(input())
   ❷ if numOfGuests:
        ❸ print('Be sure to have enough room for all your guests.')
print('Done')
```

View the execution at:
https://autbor.com/howmanyguests/.

# “TRUTHY” AND “FALSEY” VALUES

- If the user enters a blank string for name, then the while statement’s condition will be True ❶,

and program continues to ask for a name.

- If the value for numOfGuests is not 0 ❷, then the condition is considered to be True, & program will print a reminder for user ❸.

```
name=“’
❶ while not name:
        print('Enter your name:')
        name = input()
    print('How many guests will you
        have?')
    numOfGuests = int(input())
    ❷ if numOfGuests:
        ❸ print('Be sure to have enough
        room for all your guests.')
print('Done')
```

## "TRUTHY" AND "FALSEY" VALUES

- You could have entered name != '' instead of not name,
- numOfGuests != 0 instead of numOfGuests, but using the truthy and falsey values can make your code easier to read.

```
name=''
❶ while not name:
        print('Enter your name:')
        name = input()
    print('How many guests will you
        have?')
    numOfGuests = int(input())
    ❷ if numOfGuests:
        ❸ print('Be sure to have enough
        room for all your guests.')
print('Done')
```

## for Loops and the range() function

- The while loop keeps looping while its condition is True (which is the reason for its name).

- What if you want to execute a block of code only a certain number of times? ???????

- You can do this with a for loop statement and the range() function.

# for Loops and the range() function

- In code, for statement looks like:

```
for i in range(5):
```

- includes the following:
  - ✓ for keyword
  - ✓ A variable name
  - ✓ in keyword
  - ✓ A call to the range() method with up to three integers passed to it
  - ✓ A colon
  - ✓ Starting on the next line, an indented block of code (=for clause)

# for Loops and the range() Function

Consider a program - fiveTimes.py

```
print('My name is')
for i in range(5):
        print('Jimmy Five Times (' + str(i) + ')')
```

The code in the for loop's clause is run five times.

- 1.The first time it is run, the variable i is set to 0; i=0
- For clause is executed: calls print() function -will print  Jimmy Five Times (0)
- After Python finishes for loop's clause of first iteration, the execution goes back

to the top of the loop, and

- 2. for statement increments i by one. i=1
- For clause is executed: calls print() function -will print  Jimmy Five Times (1)
- After Python finishes for loop's clause of second  iteration, the execution goes back

to the top of the loop, and

- 3. for statement increments i by one. i=2

# for Loops and the range() Function

Consider a program - fiveTimes.py

```
print('My name is')
for i in range(5):
        print('Jimmy Five Times (' + str(i) + ')')
```

The code in the for loop's clause is run five times.

- 1.The first time it is run, i=0;   print  Jimmy Five Times (0)
- 2. for statement increments i by one. i=1   will print  Jimmy Five Times (1)
- 3. for statement increments i by one. i=2 :   will print  Jimmy Five Times (2)
- 4. for statement increments i by one. i=3; will print  Jimmy Five Times (3)
- 4. for statement increments i by one. i=4; will print  Jimmy Five Times (4)

This is why range(5) results in five iterations through the clause,
with i =0, then 1, then 2, then 3, and then 4.
Imp: The variable i will go up to, but will not include, the integer passed to range().

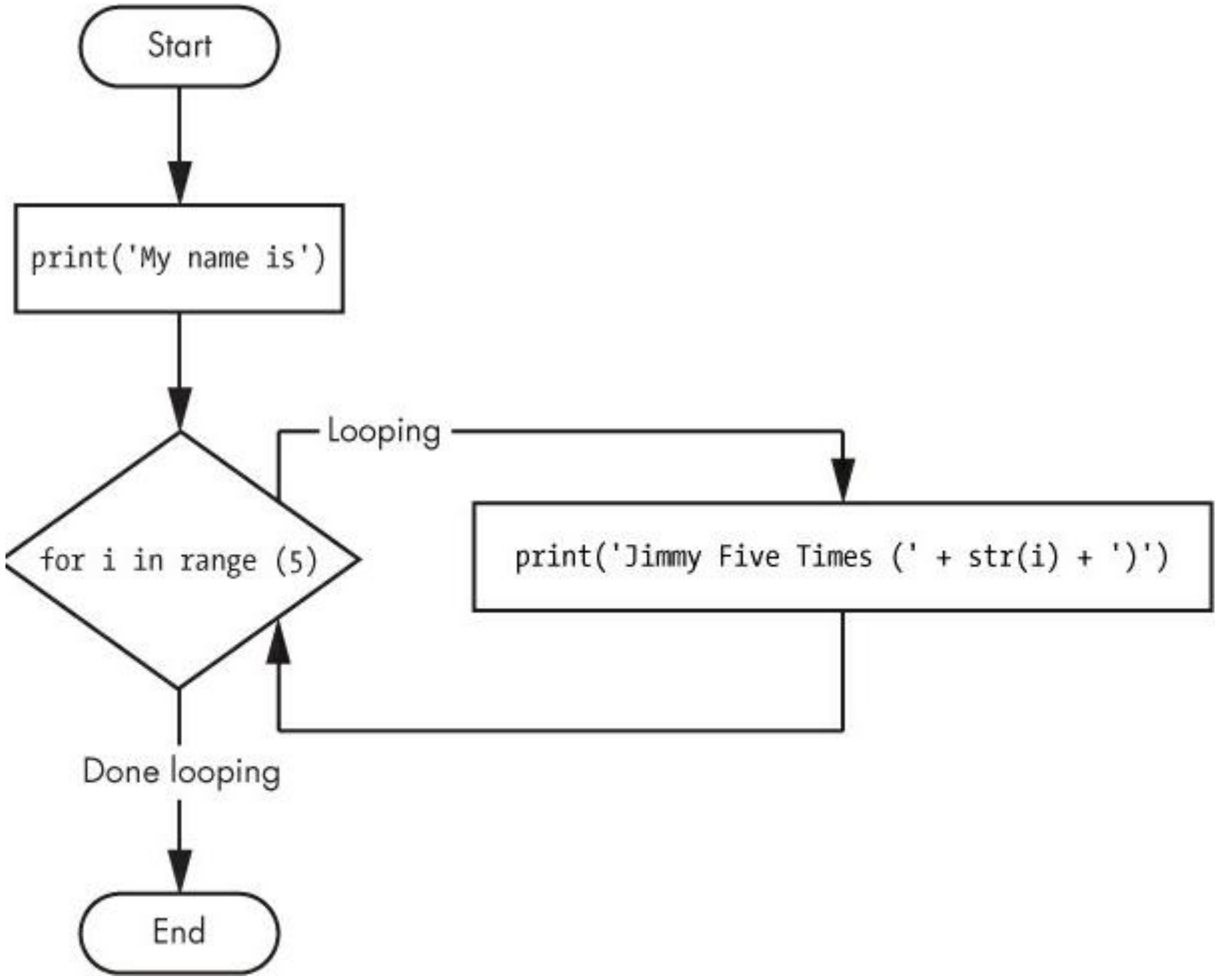
# for Loops and the range() Function

- When you run this program, it should print:

My name is
Jimmy Five Times (value of i )
5 times before leaving for loop.

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

The flowchart for fiveTimes.py



# for Loops and the range() Function

NOTE:

- for loops allow break and continue statements as well.
- The continue statement will continue to the next value of the for loop’s counter, as if the program execution had reached the end of the loop and returned to the start.

- Use continue and break statements only inside while and for loops.
- If you try to use these statements elsewhere, Python will give you an error.

# for Loops and the range() Function

Example2: to add up all the numbers from 0 to 100, Using for loop.

```
❶ total = 0
❷ for num in range(101):
        ❸ total = total + num
❹ print(total)
```

- When the program first starts, the total variable is set to 0 ❶.
- The for loop ❷ then executes total = total + num ❸ 100 times.
- By the time the loop has finished all of its 100 iterations, every integer from 0 to 100 will have been added to total. At this point, total is printed to the screen ❹.
- The result should be 5,050.
- On slowest computers, this program takes less than a second to complete.

## An Equivalent while Loop

Rewrite fiveTimes.py to use a while loop equivalent of a for loop.

```
print('My name is')
i = 0
while i < 5:
        print('Jimmy Five Times (' + str(i) + ')')
i = i + 1
```

O/P should look same as fiveTimes.py program, which uses a for loop.

View the execution of this program at https://autbor.com/fivetimeswhile/.

# Starting, Stopping, and Stepping Arguments to range()

Some functions can be called with multiple arguments separated by a Comma
Ex: range().

```
for i in range(start, stop):
        print(i)
```

- This lets the integer passed to range() to follow any sequence of integers, including starting at a number other than zero.
- start= loop's variable starts at this.   (1st argument)

end= number to stop at, Imp: will be up to, but not including(2nd argument)

## Starting, Stopping, and Stepping Arguments to range()

```
for i in range(start, stop):
        print(i)
```

```
for i in range(12, 16):
        print(i)
```

O/P:

```
12
13
14
15
```

# Starting, Stopping, and Stepping Arguments to range()

- The range() function can also be called with three arguments.
- The first two arguments will be the start and stop values, and the third will be the step argument.
- step is the amount that variable is increased by after each iteration.
- Syntax:

```
for i in range(start, stop):
        print(i)

for i in range(0, 10, 2):
        print(i)
```

O/P:

```
0
2
4
6
8
```

# Starting, Stopping, and Stepping Arguments to range()

for loop takes a negative number
then for loop counts down instead of up.

Ex:

```
for i in range(5, -1, -1):
        print(i)
```

O/P:

```
5
4
3
2
1
0
```

# Importing Modules

built-in functions= basic set of functions.
Ex: print(), input(), and len() functions

Python also comes with a set of modules called standard library.

Each module is a Python program that contains a related group of functions that can be embedded in programs

Ex: 1. math module has mathematics-related functions,
2. random module has random number-related functions, and so on

import the module with an import statement to use the functions in a module

## Importing Modules

import statement consists of following:

- ✓The import keyword
- ✓The name of the module
- ✓Sometimes, more module names, separated by commas

Once you import a module, you can use all the functions of that module.

Ex: import random module, to get access to random.randint() function.

# Importing Modules

Consider the program: printRandom.py:

```
import random
for i in range(5):
        print(random.randint(1, 10))
```

O/P:

```
4
1
8
4
1
```

view the execution of this program at
https://autbor.com/printrandom/.

# DON'T OVERWRITE MODULE NAMES

- Don't use the names of the Python's modules to save Python scripts, Ex: don't use random.py, sys.py, os.py, or math.py.
- If accidentally name is given, ex: random.py,
  and use an import random statement in another program, your program would import your random.py file instead of
  Python's random module.
- This leads to errors such as AttributeError: module 'random' has no attribute 'randint', since your random.py doesn't have the functions that real random module has.
- Don't use the names of any built-in Python functions either, such as print() or input().

## Importing Modules

- The random.randint() function:l evaluates to a random integer value between the two integers that you pass it.

- Since randint() is in the random module, first type random. in front of the function name to tell Python to look for this function inside the random module.

- Syntax:

```
import random, sys, os, math
```

## from import Statements

An alternative form of the import statement:

- ✓ from keyword,
- ✓ then the module name,
- ✓ the import keyword,
- ✓ and a star;
- ✓ Ex: from random import *

- ➢ With this form of import statement, calls to functions in random will not need the random prefix.
- ➢ But to make it readable code, so it is better to use the import random form of the statement

# Ending a program early with the sys.exit() function

how to terminate the program before last instruction???????

Programs always terminate when the execution reaches the bottom of the instructions.

sys.exit() function: when called it causes the program to terminate, or exit, before the last instruction.

import sys before using it.

Ex: exitExample.py:

# Ending a program early with the sys.exit() function

Ex: exitExample.py:

```
import sys
while True:
        print('Type exit to exit.')
        response = input()
        if response == 'exit':
                sys.exit()
        print('You typed ' + response + '.')
```

This program has an infinite loop with no break statement inside.
The only way this program will end is if the execution reaches the sys.exit() call.
When response is equal to exit, the line containing the sys.exit() call is executed.
Since the response variable is set by the input() function, the user must enter exit in order to stop the program.

# Functions

- built-in functions: print(), input(), and len() functions
- function: is a miniprogram within a program.
- Ex: Create a program helloFunc.py
- ❶ def hello():

  ```
      ❷ print('Howdy!')
      print('Howdy!!!')
              print('Hello there.')
  ```
- ❸ hello()

  ```
      hello()
      hello()
  ```

view at:
https://autbor.com/hellofunc/

# Functions

- ❶ def hello():

  ❷ print('Howdy!')

  print('Howdy!!!')

  print('Hello there.')

- ❸ hello()

  hello()

  hello()

The first line is a def statement ❶, which defines a function named hello().

code in the block that follows the def statement ❷ is the body of function

This code is executed when the function is called, not when the function is first defined

The hello() lines after the function ❸ are function calls.

# Functions

- ❶ def hello():

  ❷ print('Howdy!')

  print('Howdy!!!')

  print('Hello there.')

- ❸ hello()

  hello()

  hello()

In code, a function call is just the function's name followed by parentheses, sometimes with some number of arguments in between the parentheses.

When program execution reaches these calls, it will jump to the top line in function and begin executing code there.

When it reaches end of function, execution returns to the line that
called the function and continues moving through the code as before.

# Functions

- ❶ def hello():

  ❷ print('Howdy!')

  print('Howdy!!!')

  print('Hello there.')

- ❸ hello()

  hello()

  hello()

Since this program calls hello() three times, the code in the hello() function is executed three times.

Output:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

## Functions

- ➢ function: is a miniprogram within a program.
- ➢ Purpose of function is to group code that gets executed multiple times.

With function

```
• ❶ def hello():
         ❷  print('Howdy!')
                 print('Howdy!!!')
                                print('Hello
there.')
• ❸ hello()
         hello()
         hello()
```

Without function:

```
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
```

## Functions

- Avoid duplicating code, Why?????
- because if you ever decide to update the code—if, for example, you find a bug you need to fix—you'll have to remember to change the code everywhere you copied it.

- Deduplication makes programs shorter, easier to read, and easier to update.

## Functions

- function: is a miniprogram within a program
- Purpose of function is to group code that gets executed multiple times.

- This avoids duplication.
- Deduplication makes programs shorter, easier to read, and easier to update.

## def Statements with Parameters

- arguments: values that are passed by typing them between the parentheses when the built-in (Ex: print() or len()) or user defined function is called.

- Create a program: helloFunc2.py

```
❶ def hello(name):
      ❷ print('Hello, ' + name)
❸ hello('Alice')
  hello('Bob')
```

Output:

```
Hello, Alice
Hello, Bob
```

view at:
https://autbor.com/hellofunc2/

## def Statements with Parameters

- What are Parameters ?
- Parameters are the variables that contain arguments.
- These parameters appear between the parentheses in function definition.
- When a function is called with arguments, the arguments are stored in parameters in function definition.

```
❶ def hello(name):
       ❷ print('Hello, ' + name)
❸ hello('Alice')
   hello('Bob')
```

The definition of the hello() function in this program has a parameter called name ❶.

## def Statements with Parameters

- **Note**: the value stored in a parameter is forgotten when the function returns.
- Ex: if you add print(name) after hello('Bob') in the previous program, the program would give a NameError because there is no variable called name.
- This variable is destroyed after the function call hello('Bob') returns, so print(name) would refer to a name variable that does not exist.

```
❶ def hello(name):
    ❷ print('Hello, ' + name)
❸ hello('Alice')
  hello('Bob')
```

This is similar to how a program’s variables are forgotten when the program terminates.

# Define, Call, Pass, Argument, Parameter

**What is defining a function?**

- To define a function is to create it, just like an assignment statement like spam = 42 creates the spam variable.
- The def statement defines(=creates) the sayHello() function ❶.

The sayHello('Al') line ❷ calls the now-created function, sending the execution to the definition of the function’s code.

Code Ex:

```
❶ def sayHello(name):
        print('Hello, ' + name)
❷ sayHello('Al')
```

This function call is also known as passing the string value 'Al' to the function.

Variables that have arguments assigned to them are parameters.

# Return Values and return Statements

- When you call len() function and pass it an argument such as 'Hello', the function call evaluates to the integer value 5, which is the length of the string passed to it.

- When creating a function using the def statement, you can specify what the return value should be with a return statement.

- the value that a function call evaluates to is called the return value of the function

- A return statement consists of following
  - ✓The return keyword
  - ✓The value or expression that the function should return

## Return Values and return Statements

- When an expression is used with a return statement, the return value =value, expression evaluates to.

  - Ex:

  Consider a program, magic8Ball.py that defines a function that returns a different string depending on what number it is passed as an argument.

## Return Values and return Statements

```
➊ import random
➋ def getAnswer(answerNumber):
    ➌ if answerNumber == 1:
      return 'It is certain'
      elif answerNumber == 2:
      return 'It is decidedly so'
      elif answerNumber == 3:
      return 'Yes'
      elif answerNumber == 4:
      return 'Reply hazy try again'
      elif answerNumber == 5:
      return 'Ask again later'
      elif answerNumber == 6:
return 'Concentrate and ask again'
elif answerNumber == 7:
return 'My reply is no'
elif answerNumber == 8:
return 'Outlook not so good'
elif answerNumber == 9:
return 'Very doubtful'
➍ r = random.randint(1, 9)
➎ fortune = getAnswer(r)
➏ print(fortune)
```

## Return Values and return Statements

```
❶ import random
❷ def getAnswer(answerNumber):
     ❸ if answerNumber == 1:
        return 'It is certain'
        elif answerNumber == 2:
        return 'It is decidedly so'
        elif answerNumber == 3:
        return 'Yes'
        elif answerNumber == 4:
        return 'Reply hazy try again'
        elif answerNumber == 5:
        return 'Ask again later'
        elif answerNumber == 6:
```

Python imports the random module ❶.
Then getAnswer() function is defined ❷

Because the function is being defined (and not called), the execution skips over the code in it.

## Return Values and return Statements

```
return 'Concentrate and ask again'
elif answerNumber == 7:
return 'My reply is no'
elif answerNumber == 8:
return 'Outlook not so good'
elif answerNumber == 9:
return 'Very doubtful'
❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

Next, the random.randint() function is called with two arguments: 1 and 9 ❹ It evaluates to a random integer between 1 and 9 (including 1 and 9 themselves), and this value is stored in a variable named r.

# Return Values and return Statements

```
❶ import random
❷ def getAnswer(answerNumber):
    ❸ if answerNumber == 1:
       return 'It is certain'

return 'Concentrate and ask again'
elif answerNumber == 7:
return 'My reply is no'
elif answerNumber == 8:
return 'Outlook not so good'
elif answerNumber == 9:
return 'Very doubtful'
❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

- The getAnswer() function is called with r as the argument ❺.
- The program execution moves to the top of the getAnswer() function ❸, and the value r is stored in a parameter named answerNumber.
- Then, depending on the value in answerNumber, the function returns one of many possible string values.
- The program execution returns to the line at the bottom of the program that originally called getAnswer() ❺.
- The returned string is assigned to a variable named fortune, which then gets passed to a print() call ❻ and is printed to the screen.

# Return Values and return Statements

Note:

you can pass return values as an argument to another function call, you could shorten these three lines as:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

expressions are composed of values and operators.
A function call can be used in an expression as value because the call evaluates to its return value.

single equivalent line:

```
print(getAnswer(random.randint(1, 9)))
```

## The None Value

- In Python, there is a value called None, which represents the absence of a value.

- The None value is the only value of the NoneType data type.

- (Other programming languages might call this value null, nil, or undefined.)

- Just like the Boolean True and False values, None must be typed with a capital N.

- This value-without-a-value can be helpful when you need to store something that won’t be confused for a real value in a variable.

## The None Value

- One place where None is used is as the return value of print().

- The print() function displays text on the screen, but it doesn't need to return anything in the same way len() or input() does.

- But since all function calls need to evaluate to a return value, print() returns None.

# The None Value

- >>> spam = print('Hello!')

Hello!

- >>> None == spam

True

Behind the scenes, Python adds return None to the end of any function definition with no return statement.

Also, if you use a return statement without a value (that is, just the return keyword by itself), then None is returned.

This is similar to how a while or for loop implicitly ends with a continue statement.

## Keyword Arguments and the print() Function

- Most arguments are identified by their position in function call.
- Ex: random.randint(1, 10) ≠ random.randint(10, 1).
- The function call random.randint(1, 10) will return a random integer between 1 and 10 because the first argument is the low end of the range and the second argument is the high end.
- while random.randint(10, 1) causes an error
- rather than through their position, keyword arguments are identified by the keyword put before them in the function call. Keyword arguments are often used for optional parameters.

# Keyword Arguments and the print() Function

- rather than through their position, keyword arguments are identified by the keyword put before them in the function call. Keyword arguments are often used for optional parameters.

- Ex: the print() function has the optional parameters end and sep to specify what should be printed at end of its arguments and between its arguments (separating them), respectively.

- Program:

```
print('Hello')
print('World')
```

Output:

```
Hello
World
```

# Keyword Arguments and the print() Function

- Program:

```
print('Hello')
print('World')
```

Output:

```
Hello
World
```

- The two outputted strings appear on separate lines because the print() function automatically adds a newline character to the end of the string it is passed.

- You can set the end keyword argument to change the newline character to a different string.

- Ex:

```
print('Hello', end='')
print('World')
```

Output:

```
HelloWorld
```

# Keyword Arguments and the print() Function

- Program:

```
print('Hello')
print('World')
```

Output:

```
Hello
World
```

- Ex:

```
print('Hello', end='')
print('World')
```

Output:

```
HelloWorld
```

- The output is printed on a single line because there is no longer a newline printed after 'Hello'.
- Instead, the blank string is printed.
- This is useful if you need to disable the newline that gets added to the end of every print() function call.

# Keyword Arguments and the print() Function

Similarly, when you pass multiple string values to print(), the function will automatically separate them with a single space.

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

But you could replace the default separating string by passing the sep keyword argument a different string.

Ex:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

Takeaway: some functions have optional keyword arguments that can be specified when the function is called.

## Local and Global Scope

Local Scope: Parameters and variables that are assigned in a called function are said to exist in that function's local scope.

- A variable that exists in a local scope is called a local variable.
- A local scope is created whenever a function is called.
- Any variables assigned in function exist within function's local scope.
- When the function returns, the local scope is destroyed, and these variables are forgotten.
- The next time you call the function, the local variables will not

remember values stored in them from the last time the function was called.

Global Scope: Variables that are assigned outside all functions are said to exist in the global scope.

- A variable that exists in the global scope is called a global variable.
- There is only one global scope, & it is created when program begins
- When program terminates, the global scope is destroyed,

  and all its variables are forgotten.

# Local and Global Scope

A variable must be one or the other; It cannot be both local & global.

Scopes matter for several reasons:

- Code in the global scope, outside of all functions, cannot use any local variables.
- However, code in a local scope can access global variables. Code in a function’s local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

# Local and Global Scope

Python has different scopes instead of just making everything a global variable. Why?

- ➢ when variables are modified by the code in a particular call to a function, the function interacts with the rest of the program only through its parameters and the return value.
- ➢ This narrows down number of lines of code that may be causing a bug.
- ➢ If your program contained nothing but global variables and had a bug because of a variable being set to a bad value, then it would be hard to track down where this bad value was set.
- ➢ It could have been set from anywhere in the program, and your program could be hundreds or thousands of lines long!
- ➢ But if the bug is caused by a local variable with a bad value, you know that only the code in that one function could have set it incorrectly.
- ➢ While using global variables in small programs is fine, it is a bad habit to rely on global variables as programs get larger and larger.

# Local Variables Cannot Be Used in Global Scope

Consider the program:

```
def spam():
  ➊ eggs = 31337
spam()
print(eggs)
```

Output

```
Traceback (most recent call last):
File "C:/test1.py", line 4, in <module> print(eggs)
NameError: name 'eggs' is not defined
```

- ➢ The error happens because the eggs variable exists only in the local scope which got created when spam() is called ➊.
- ➢ Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs.
- ➢ So when your program tries to run print(eggs), Python gives an error saying that eggs is not defined.
- ➢ When the program execution is in the global scope, no local scopes exist, so there can’t be any local variables.
- ➢ This is why only global variables can be used in the global scope.

# Local Scopes Cannot Use Variables in Other Local Scopes

- A new local scope is created whenever a function is called, including when a function is called from another function.

- Consider this program:

```
def spam():
        ❶ eggs = 99
        ❷ bacon()
        ❸ print(eggs)
def bacon():
          ham = 101
        ❹ eggs = 0
❺ spam()
```

Footnote ; Multiple local scopes can exist at the same time.

- When the program starts, the spam() function is called ❺, and a local scope is created.
- The local variable eggs ❶ is set to 99.
- Then the bacon() function is called ❷, and a second local scope is created.
- In this new local scope, local variable ham is set to 101, & a local variable eggs—which is different from one in spam()'s local scope—is also created ❹ & set to 0.
- When bacon() returns, the local scope for that call is destroyed, including its eggs variable. The program execution continues in the spam() function to print the value of eggs ❸.
- Since local scope for call to spam() still exists, the only eggs variable is the spam() function's eggs variable, which was set to 99. This is what program prints.

## Global Variables Can Be Read from a Local Scope

Consider the following program:

```
def spam():
        print(eggs)
eggs = 42
spam()
print(eggs)
```

- ➢ Since there is no parameter named eggs or any code that assigns eggs a value in the spam() function, when eggs is used in spam(), Python considers it a reference to the global variable eggs.
- ➢ This is why 42 is printed when the previous program is run.

## Local and Global Variables with the Same Name

Technically, it’s perfectly acceptable to use the same variable name for a global variable and local variables in different scopes in Python.
But, better avoid doing this, to make job simpler.
Consider a program, localGlobalSameName.py:

```
def spam():
        ❶ eggs = 'spam local'
        print(eggs) # prints 'spam local'
def bacon():
        ❷ eggs = 'bacon local'
        print(eggs) # prints 'bacon local'
        spam()
        print(eggs) # prints 'bacon local'
❸ eggs = 'global'
bacon()
print(eggs) # prints 'global'
```

Outputs:

```
bacon local
spam local
bacon local
global
```

## Local and Global Variables with the Same Name

There are actually three different variables in this program, but confusingly they are all named eggs. The variables are as follows:

❶ A variable named eggs that exists in a local scope when spam() is called.

❷ A variable named eggs that exists in a local scope when bacon() is called.

❸ A variable named eggs that exists in the global scope.

```
def spam():
        ❶ eggs = 'spam local'
        print(eggs) # prints 'spam local'
def bacon():
        ❷ eggs = 'bacon local'
        print(eggs) # prints 'bacon local'
        spam()
        print(eggs) # prints 'bacon local'
❸ eggs = 'global'
bacon()
print(eggs) # prints 'global'
```

Outputs:

```
bacon local
spam local
bacon local
global
```

- Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time.
- This is why you should avoid using the same variable name in different scopes.

## The global Statement

- If you need to modify a global variable from within a function, use the global statement.
- If you have a line such as global variablename (Ex:global eggs) at the top of a function, it tells Python, “In this function, eggs refers to global variable, so don’t create a local variable with this name.”
- Ex: consider a program, globalStatement.py:

```
def spam():
        ➊ global eggs
        ➋ eggs = 'spam'
eggs = 'global'
spam()
print(eggs)
```

Output:

```
spam
```

Because eggs is declared global at the top of spam() ❶, when eggs is set to 'spam' ❷, this assignment is done to the globally scoped eggs. No local eggs variable is created.

## The global Statement

There are four rules to tell whether a variable is in a local scope or global scope:

- If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
- If there is a global statement for that variable in a function, it is a global variable.
- Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
- But if the variable is not used in an assignment statement, it is a global variable.

- Example:

## The global Statement

There are four rules to tell whether a variable is in a local scope or global scope:

- If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
- If there is a global statement for that variable in a function, it is a global variable.
- Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
- But if the variable is not used in an assignment statement, it is a global variable.

- Ex: Consider a program, sameNameLocalGlobal.py::

## The global Statement: sameNameLocalGlobal.py:

```
def spam():
        ❶ global eggs
        eggs = 'spam' # this is the global
def bacon():
        ❷ eggs = 'bacon' # this is a local
def ham():
        ❸ print(eggs) # this is the global
eggs = 42 # this is the global
spam()
print(eggs)
```

Output:

```
spam
```

Imp: In a function, a variable will either always be global or always be local. The code in a function can’t use a local variable named eggs and then use the global eggs variable later in that same function.

In the spam() function, eggs is the global eggs variable because there’s a global statement for eggs at the beginning of the function ❶.
In bacon(), eggs is a local variable because there’s an assignment statement for it in that function ❷.
In ham() ❸, eggs is the global variable because there is no assignment statement or global statement for it in that function.

# The global Statement

NOTE: If you ever want to modify the value stored in a global variable from in a function, you must use a global statement on that variable.

- The code in a function can’t use a local variable named eggs and then use the global eggs variable later in that same function.
- If you try to use a local variable in a function before you assign a value

to it, Python will give you an error. Consider a program, sameNameError.py:

```
def spam():
        print(eggs)            # ERROR!
        ❶ eggs = 'spam local'
❷ eggs = 'global'
spam()
```

it produces an error message:

```
Traceback (most recent call last):
File "C:/sameNameError.py", line 6, in
<module> spam()
File "C:/sameNameError.py", line 2, in spam
print(eggs) # ERROR!
UnboundLocalError: local variable 'eggs'
referenced before assignment
```

# The global Statement

Consider a program, sameNameError.py:

```
def spam():
        print(eggs)            # ERROR!
        ❶ eggs = 'spam local'
❷ eggs = 'global'
spam()
```

This error happens because

- ✓ an assignment statement for eggs in spam() function ❶ identified.
- ✓ Hence considers eggs to be local.
- ✓ print(eggs) is executed before eggs is assigned anything,
- ✓ the local variable eggs doesn’t exist.
- ✓ global eggs variable ❷ will not be considered.

it produces an error message:

```
Traceback (most recent call last):
File "C:/sameNameError.py", line 6, in
<module> spam()
File "C:/sameNameError.py", line 2, in spam
print(eggs) # ERROR!
UnboundLocalError: local variable 'eggs'
referenced before assignment
```

## Exception (=Error) Handling

- If an error (=exception), is encountered, Python program will crash
- This needs to be avoided in real world programs.
- Instead, the program should detect errors, handle them, and then continue to run.

Ex: consider the following program, zeroDivide.py:

```
def spam(divideBy):
        return 42 / divideBy
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

- Defined a function called spam,
- given it a parameter, and then
- printed the value of that function with various parameters to see what happens.

# Exception (=Error) Handling

zeroDivide.py:

```
def spam(divideBy):
        return 42 / divideBy
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Output:

```
21.0
3.5
Traceback (most recent call last):
File "C:/zeroDivide.py", line 6, in <module>
print(spam(0))
File "C:/zeroDivide.py", line 2, in spam
return 42 / divideBy
ZeroDivisionError: division by zero
```

A ZeroDivisionError happens whenever you try to divide a number by zero.

From the line number given in the error message, you know that the return statement in spam() is causing an error.

# Exception (=Error) Handling

- Errors can be handled with try and except statements.
- The code that could potentially have an error is put in a try clause.
- The program execution moves to the start of a following except clause if an error happens.

put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

When code in a try clause causes an error, the program execution immediately moves to the code in the except clause.
After running that code, the execution continues as normal.

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

# Exception (=Error) Handling

When code in a try clause causes an error, the program execution immediately moves to the code in the except clause.
After running that code, the execution continues as normal.

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Output:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

## Exception (=Error) Handling

- Any errors that occur in function calls in a try block will also be caught.
- Consider the program: which has the spam() calls in the try block:

```
def spam(divideBy):
    return 42 / divideBy
try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
print('Error: Invalid argument.')
```

Output:

```
21.0
3.5
Error: Invalid argument.
```

The reason print(spam(1)) is never executed: because once the execution jumps to the code in the except clause, it does not return to the try clause.
Instead, it just continues moving down the program as normal.

# A short Program: guess the number.

```
# This is a guess the number game.

import random      #to generate random number

secretnumber=random.randint(1,20) print('i am thinking
a number between 1 and 20 ')

# Ask the player to guess 6 times.
for guessestaken in range(1,7):
print('Take a guess')
   guess=int(input())
   if guess<secretnumber:
      print('your guess is too low')
   elif guess> secretnumber:
      print('your guess is too high')
   else:
      break      #this is the correct guess case
```

```
if guess==secretnumber:
   print('Good job! You guessed the secret
number i thought. for this you have taken
'+str(guessestaken)+' guesses')
else:
   print('Nope. Number i was thinking of was  ' +
str(secretnumber))
```

# A short Program: guess the number.

```
# This is a guess the number game.

import random      #to generate random number

secretnumber=random.randint(1,20)    #random nember generated
between 1, 20 (both inclusive)
print('i am thinking a number between 1 and 20 ')

# Ask the player to guess 6 times.
for guessestaken in range(1,7):    #Change this range and check
    print('Take a guess')
    guess=int(input())
    if guess<secretnumber:
        print('your guess is too low')
    elif guess> secretnumber:
        print('your guess is too high')
    else:
        break      #this is the correct guess case

if guess==secretnumber:
    print('Good job! You guessed the secret number i thought. for this
you have taken  '+str(guessestaken)+' guesses')
else:
    print('Nope. Number i was thinking of was  ' + str(secretnumber))
```

**Output:**

```
I am thinking a number
between 1 and 20 Take a guess
10
your guess is too high
Take a guess
5
Good job! You guessed the
secret number i thought. for
this you have taken 2 guesses
```