Python Programming

NSC Workshop (November 2024)

Python Programming

Natworpong Loyswai (Peng)
natworpong@shipsgp.com

Boonnithi Jiaramaneepinit (Up)
shiragap@shipsgp.com

Sakulthip Rassameecharerntham (Yok)
peony@shipsgp.com

Agenda

- Python: Getting Started
- Python: Fundamentals
  - Arithmetic Operations
  - Fundamental Data Types
  - Conditional Statements
  - Loops
  - Function and Error Handling
- Introduction to Object-Oriented Programming: Classes
- Hands-On Exercise 1: Coding Fundamentals 1

Agenda

- Advanced Python Utilities
  - Modules
  - Imports
  - File Handling
- Hands-On Exercise 2: Coding Fundamentals 2
  - Module Maven
  - Error Expert
  - File Fanatic

Python: Getting Started

Visual Studio Code

https://code.visualstudio.com/Download

Python

https://www.python.org/downloads/

Arithmetic Operations

Arithmetic Operations

Name	*Meaning*	*Example*	*Result*
`+`	Addition	`34 + 1`	`35`
`-`	Subtraction	`34.0 - 0.1`	`33.9`
`*`	Multiplication	`300 * 30`	`9000`
`/`	Float Division	`1 / 2`	`0.5`
`//`	Integer Division	`1 // 2`	`0`
`**`	Exponentiation	`4 ** 0.5`	`2.0`
`%`	Remainder	`20 % 3`	`2`

Arithmetic Operations

```
>>> print(2 + 5)
7
>>> print(5 % 2)
1
>>> print((5 + 2) * 3)
21
>>> print(10 / 5 + 1 * 4 - 2 * 2)
2
>>> print(5 // 2)
2
>>> 1 + 1 + 1 + 1 * 10 + 1
14
>>> |
```

Fundamental Data Types

Fundamental Data Types

- **Numbers**

```
b = 4.5                          # Floating point
c = 517288833333L   # Long integer (arbitrary precision)
d = 4 + 3j                       # Complex (imaginary) number
```

- **Strings**

```
b = "World"                                          # Double quotes
c = "Bob said 'hey there.'"   # A mix of both
d = '''A triple quoted string can span multiple lines
like this'''
e = """Also works for double quotes"""
```

Fundamental Data Types

- **Lists of Arbitrary Objects**

```
a = [2, 3, 4]                    # A list of integers
b = [2, 7, 3.5, "Hello"]         # A mixed list
c = []                           # An empty list
d = [2, [a,b]]                   # A list containing a list
e = a + b                        # Join two lists
```

- **List Manipulation**

```
x = a[1]                         # Get 2nd element (0 is first)
y = b[1:3]                       # Return a sublist
z = d[1][0][2]                   # Nested lists
b[0] = 42                        # Change an element
```

Basic Types (Tuples)

Tuples

```
f = (2,3,4,5)                                   # A tuple of integers # An empty tuple
g = (,)                                         # A tuple containing mixed objects
h = (2, [3,4], (10,11,12))
```

Tuple Manipulation

```
x = f[1]                                        # Element access. x = 3 # Slices. y = (3,4)
y = f[1:3]                                      # Nesting. z = 4
z = h[1][1]
```

Comments

- Tuples are like lists, but size is fixed at time of creation. Can’t replace members (said
- to be "immutable")

Basic Types (Dictionaries)

Dictionaries (Associative Arrays)

```
a = { }                                  # An empty dictionary
b = { 'x': 3, 'y': 4 }
c = { 'uid': 105,
      'login': 'beazley', 'name' : 'David Beazley'
    }
```

Dictionary Access

```
u = c['uid'] c['shell'] = "/bin/sh"          # Get an element # Set an element
if c.has_key("directory"): d = c['directory']  # Check for presence of an member
else:
    d = None

d = c.get("directory",None)                  # Same thing, more compact
```

Formatting

Frequently used specifiers

Specifier	*Format*
`"10.2f"`	Format the float item with width 10 and precision 2.
`"10.2e"`	Format the float item in scientific notation with width 10 and precision 2.
`"5d"`	Format the integer item in decimal with width 5.
`"5x"`	Format the integer item in hexadecimal with width 5.
`"5o"`	Format the integer item in octal with width 5.
`"5b"`	Format the integer item in binary with width 5.
`"10.2%"`	Format the number in decimal.
`"50s"`	Format the string item with width 50.
`"<10.2f"`	Left-justify the formatted item.
`">10.2f"`	Right-justify the formatted item.

Formatting

Methods: format, round

```
>>> amount = 12618.98
>>> interestRate = 0.0013
>>> interest = amount * interestRate
>>> print("Interest is", interest)
Interest is 16.404674

>>> print("Interest is", round(interest, 2))
Interest is 16.4

>>> print("Interest is", format(interest, ".2f"))
Interest is 16.40
```

Formatting

Floating -Point Numbers: f

```
print(format(57.467657, "10.2f"))
print(format(12345678.923, "10.2f"))
print(format(57.4, "10.2f"))
print(format(57, "10.2f"))
```

displays

```
|←— 10 —→|
□□□□ 57.47
 123456782.92
□□□□ 57.40
□□□□ 57.00
```

Formatting

Scientific Notation: e

```
print(format(57.467657, "10.2e"))
print(format(0.0033923, "10.2e"))
print(format(57.4, "10.2e"))
print(format(57, "10.2e"))
```

displays

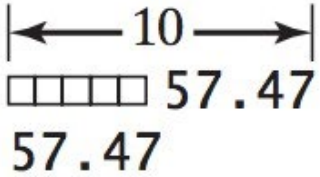
```
|←——10——→|
□□5.75e+01
□□3.39e-03
□□5.74e+01
□□5.70e+01
```

Formatting

Justifying Format: <, >

```
print(format(57.467657, "10.2f"))
print(format(57.467657, "<10.2f"))
```

displays



Formatting

Formatting Integers: d, x, o, b

```
print(format(59832, "10d"))
print(format(59832, "<10d"))
print(format(59832, "10x"))
print(format(59832, "<10x"))
```

displays

```
|←— 10 —→|
□□□□□ 59832
59832
□□□□□ e9b8
e9b8
```

Formatting

Formatting Strings: s

```
print(format("Welcome to Python", "20s"))
print(format("Welcome to Python", "<20s"))
print(format("Welcome to Python", ">20s"))
print(format("Welcome to Python and Java", ">20s"))
```

displays

```
|←———— 20 ————→|
Welcome to Python
Welcome to Python
□□□ Welcome to Python
Welcome to Python and Java
```

Conditional Statements

Boolean Expression

Boolean value: True, False

TABLE 4.1 Comparison Operators

Python Operator	*Name*	*Example* (radius is 5)	*Result*
`<`	less than	`radius < 0`	`False`
`<=`	less than or equal to	`radius <= 0`	`False`
`>`	greater than	`radius > 0`	`True`
`>=`	greater than or equal to	`radius >= 0`	`True`
`==`	equal to	`radius == 0`	`False`
`!=`	not equal to	`radius != 0`	`True`

Boolean Expression

Method: bool

```
>>> is_Student = True
>>> print(is_Student)
True
>>> int(is_Student)
1
>>>
```

```
>>> is_Student = 1
>>> is_Student
1
>>> bool(is_Student)
True
>>>
```

```
>>> is_Awake = False
>>> is_Awake
False
>>> int(is_Awake)
0
>>>
```

```
>>> is_Awake = 0
>>> is_Awake
0
>>> bool(is_Awake)
False
>>>
```

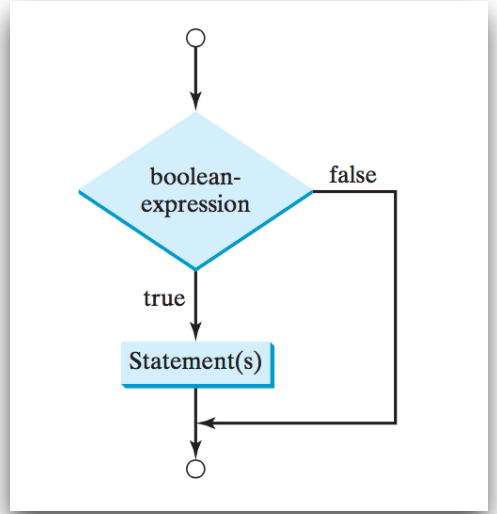
Boolean Expression

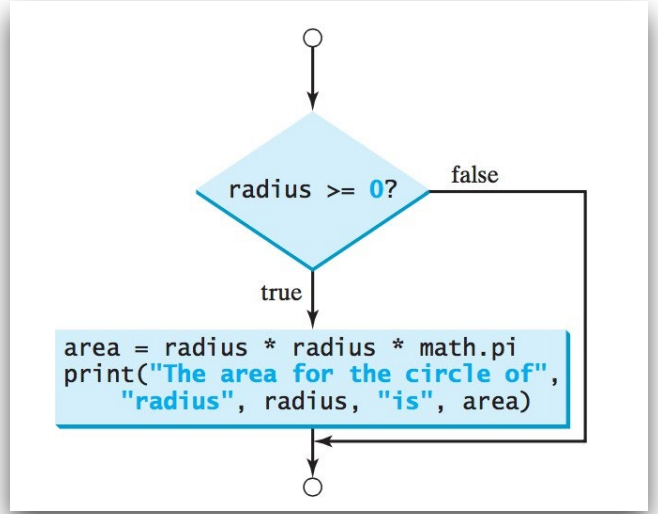
Object and Class Type

```
>>>
>>> isBool = True
>>> id(isBool)
4297020064
>>> type(isBool)
<class 'bool'>
>>>
```

if Statement

Flowchart





A one-way **if** *statement executes the statements if the condition is true.*

if Statement

Sample Task

```
number = eval(input("Enter an integer: "))

if number % 5 == 0:
    print("HiFive")

if number % 2 == 0:
    print("HiEven")
```

```
Enter an integer: 4 ↵Enter
HiEven
```

```
Enter an integer: 30 ↵Enter
HiFive
HiEven
```

Conditionals

if-else

```
# Compute maximum (z) of a and b if
a < b:
    z = b
else:
    z = a
```

The pass statement

```
if a < b: # Do nothing
      pass else: z = a
```

Notes:

Indentation used to denote bodies. pass used to denote an empty body. There is no '?:' operator.

Conditionals

elif statement

```
if a == '+':
    op = PLUS
elif a == '-':
op = MINUS
elif a == '*':
    op = MULTIPLY
else:
    op = UNKNOWN
```

Note: There is no `switch` statement.

Boolean expressions: and, or, not

```
if b >= a and b <= c:
    print "b is between a and c" if not (b
< a or b > c):
    print "b is still between a and c"
```

Loops

The *while* Loop

Syntax

```
while loop-continuation-condition:
    # Loop body
    Statement(s)
```

A `while` *loop executes statements repeatedly as long as a condition remains true.*

The *while* Loop

Syntax

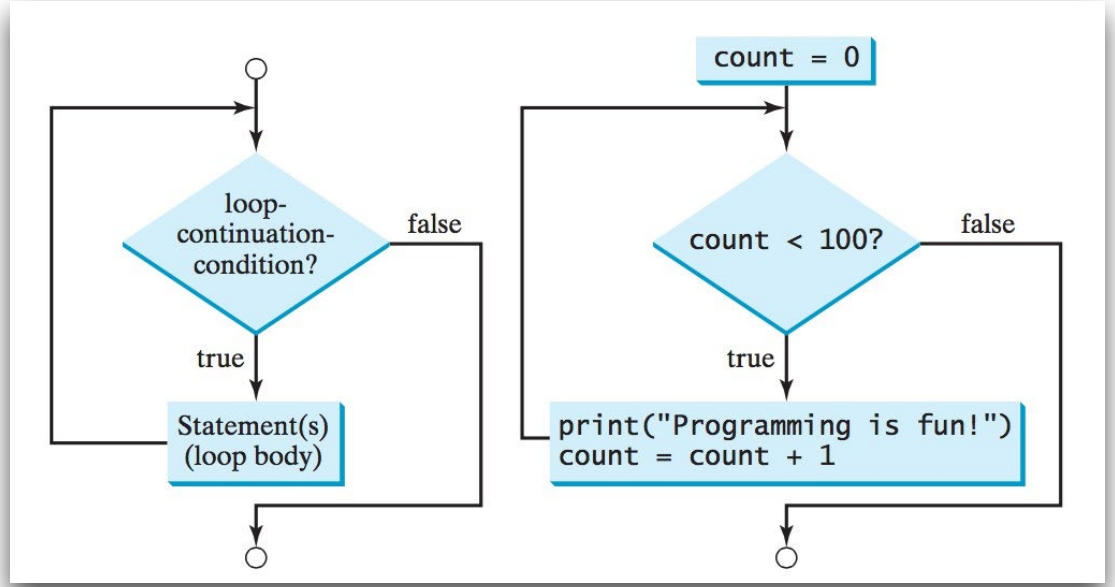
loop -continuation -condition

```
count = 0
while count < 100:
    print("Programming is fun!")
    count = count + 1
```

loop
body

The *while* Loop

Flowchart



A **`while`** *loop executes statements repeatedly as long as a condition remains true.*

The *while* Loop

Loop design strategies

1. Prepare the statements that need to be repeated
2. Wrap these statements in a loop

```
while True:
    Statements
```

3. Code the loop-continuation-condition
4. add appropriate statements for controlling the loop.

```
while loop-continuation-condition:
    Statements
    Additional statements for controlling the loop
```

The *while* Loop

!!! CAUTION !!!

Make sure your program contain **no**

INFINITE LOOP

The *while* Loop

Controlling a Loop with User Confirmation

```
continueLoop = 'Y'
while continueLoop == 'Y':
    # Execute the loop body once
    ...

    # Prompt the user for confirmation
    continueLoop = input("Enter Y to continue and N to quit: ")
```

loop -continuation -condition

Loop Controller

The *while* Loop

Controlling a Loop with a Sentinel Value

```
data = eval(input("Enter an integer (the input ends " +
    "if it is 0): "))

# Keep reading data until the input is 0
sum = 0
while data != 0:
    sum += data

    data = eval(input("Enter an integer (the input ends " +
        "if it is 0): "))

print("The sum is", sum)
```

A loop that uses a sentinel value in this way is called ***a sentinel-controlled loop.***

The *while* Loop

Controlling a Loop with a Sentinel Value

```
Enter an integer (the input ends if it is 0): 2
Enter an integer (the input ends if it is 0): 3
Enter an integer (the input ends if it is 0): 4
Enter an integer (the input ends if it is 0): 0
The sum is 9
```

A loop that uses a sentinel value in this way is called ***a sentinel-controlled loop.***

The *while* Loop

!!! CAUTION !!!

DO NOT USE

FLOATING POINT

For equality checking in a loop control

The *for* Loop

Sample

```
>>> for v in range(4, 8):
...     print(v)
...
4
5
6
7
```

```
>>> for v in range(3, 9, 2):
...     print(v)
...
3
5
7
```

The *for* Loop

Syntax

```
i = initialValue  # Initialize loop-control variable
while i < endValue:
    # Loop body
    ...
    i += 1  # Adjust loop-control variable
```

while loop

```
for i in range(initialValue, endValue):
    # Loop body
```

for loop

A Python **for** *loop iterates through each value in a sequence.*

The *for* Loop

Sample

```
>>> for v in range(5, 1, -1):
...     print(v)
...
5
4
3
2
```

The *for* Loop

!!! CAUTION !!!

The number in the range function

MUST BE

INTEGER

Nested Loops

```
print("          Multiplication Table")
# Display the number title
print("  |", end = '')
for j in range(1, 10):
    print("  ", j, end = '')
print()  # Jump to the new line
print("--------------------------------------")

# Display table body
for i in range(1, 10):
    print(i, "|", end = '')
    for j in range(1, 10):
        # Display the product and align properly
        print(format(i * j, "4d"), end = '')
    print()  # Jump to the new line
```

A loop can be nested inside another loop.

Nested Loops

```
                Multiplication Table
          1    2    3    4    5    6    7    8    9
 ------------------------------------------------------
 1 |      1    2    3    4    5    6    7    8    9
 2 |      2    4    6    8   10   12   14   16   18
 3 |      3    6    9   12   15   18   21   24   27
 4 |      4    8   12   16   20   24   28   32   36
 5 |      5   10   15   20   25   30   35   40   45
 6 |      6   12   18   24   30   36   42   48   54
 7 |      7   14   21   28   35   42   49   56   63
 8 |      8   16   24   32   40   48   56   64   72
 9 |      9   18   27   36   45   54   63   72   81
```

A loop can be nested inside another loop.

Break and continue

provide additional control to stop a loop

Break and continue

Reserved word (keyword): break

```
sum = 0
number = 0

while number < 20:
    number += 1
    sum += number
    if sum >= 100:
        break

print("The number is", number)
print("The sum is", sum)
```

Break the loop if sum is more than or equal to 100

```
The number is 14
The sum is 105
```

The **break** keyword immediately ends the innermost loop, which contains the break.

Break and continue

Reserved word (keyword): continue

```
sum = 0
number = 0

while number < 20:
    number += 1
    if number == 10 or number == 11:
        continue
    sum += number

print("The sum is", sum)
```

jump to the end of iteration
(start the loop again)

```
The sum is 189
```

The **continue** keyword ends only the current iteration.

Function Definitions and Calls

Simple Function Definition

```
def greet():
        print("Hello!")

greet()
```

```
Output:
Hello!
```

Opening Problem

Find the sum of integers from 1 to 10, from 20 to 37, and from 35 to 49, respectively.

Solution

```
sum = 0
for i in range(1, 10 + 1):
        sum += i
print("Sum from 1 to 10 is", sum)

sum = 0
for i in range(20, 37 + 1):
        sum += i
print("Sum from 20 to 37 is", sum)

sum = 0
for i in range(35, 49 + 1):
        sum += i
print("Sum from 35 to 49 is", sum)
```

Solution

```
sum = 0
for i in range(1, 10 + 1):
        sum += i
print("Sum from 1 to 10 is", sum)

sum = 0
for i in range(20, 37 + 1):
        sum += i
print("Sum from 20 to 37 is", sum)

sum = 0
for i in range(35, 49 + 1):
        sum += i
print("Sum from 35 to 49 is", sum)
```

Solution

```
def sum(i1, i2):
        result = 0
        for i in range(i1, i2 + 1):
                result += i
        return result

def main():
        print("Sum from 1 to 10 is", sum(1, 10))
        print("Sum from 20 to 37 is", sum(20, 37))
        print("Sum from 35 to 49 is", sum(35, 49))

main() # Call the main function
```

Objectives

- To define functions.
- To invoke value-returning functions.
- To invoke functions that does not return a value.
- To pass arguments by values.
- To pass arguments by values.
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain.
- To create modules for reusing functions.
- To determine the scope of variables.
- To define functions with default arguments.
- To return multiple values from a function.
- To apply the concept of function abstraction in software development.
- To design and implement functions using stepwise refinement.
- To simplify drawing programs using functions.

Defining Functions

Define a function

function name

formal parameters

function header

```
def max(num1, num2):
```

function body

```
    if num1 > num2:
        result = num1
    else:
        result = num2

    return result
```

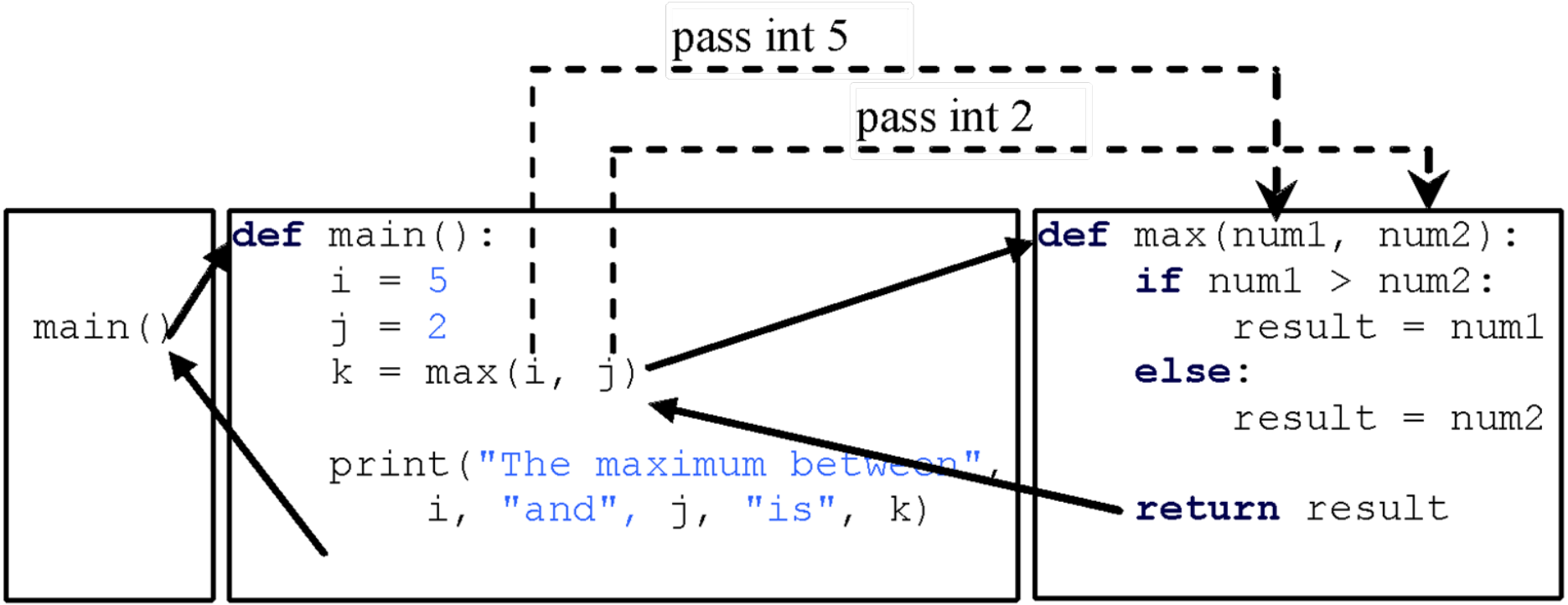
return value

Invoke a function

```
z = max(x, y)
```

actual parameters (arguments)

Calling Functions



Returning Values from Functions

```
def add_numbers(a, b):
        return a + b

result = add_numbers(3, 4)
print(result)
```

```
Output:
7
```

Pass by Value

Definition:

- A copy of the variable is passed to the function.

Characteristics:

- Original variable remains unchanged.
- Common in languages like C for primitive data types.

Pass by Value Example in Python

python Copy code

```python
def increment(number):
    number += 1
    print("Inside function:", number)

num = 10
increment(num)
print("Outside function:", num)
```

- **Output:**

  bash Copy code

  ```bash
  Inside function: 11
  Outside function: 10
  ```

- **Explanation:**
  - `number` is a copy of `num`.
  - Modifying `number` doesn't affect `num`.

Pass by Reference

Definition:

- A reference to the actual variable is passed.

Characteristics:

- Changes inside the function affect the original variable.
- Used in languages like C++ with reference parameters.

Pass by Reference Example in Python

python Copy code

```python
def add_item(my_list):
    my_list.append(4)
    print("Inside function:", my_list)

lst = [1, 2, 3]
add_item(lst)
print("Outside function:", lst)
```

- **Output:**

bash Copy code

```bash
Inside function: [1, 2, 3, 4]
Outside function: [1, 2, 3, 4]
```

- **Explanation:**
  - `my_list` refers to the same object as `lst`.
  - Modifications affect the original list.

Scope of Variables

Scope:

the part of the program where the variable can be referenced.

Scope of Variables: Example 1

```
globalVar = 1

def f1():
        localVar = 2
        print(globalVar)
        print(localVar)

f1()
print(globalVar)
print(localVar)      # Error! Out of scope.
```

Scope of Variables: Example 2

```
x = 1

def f1():
        x = 2
        print(x)        # Displays 2

f1()
print(x)                # Displays 1
```

Scope of Variables: Example 3

```
x = eval(input("Enter a number: "))

if (x > 0):
        y = 4

print(y)        # Error: y is not defined
```

Scope of Variables: Example 4

```
x = 1

def increase():
        global x
        x =  x + 1
        print(x)        # Displays 2

increase()
print(x)                # Displays 2
```

Error Handling Techniques

Programming Errors

- Syntax Error
- Run-time

Cause program to terminate abnormally

- Error Logic Error → Unexpected outcome

Programming Errors

Syntax Error : GRAMMATICAL ERROR

- Mistyping
- Indentation
- Omitting necessary punctuation
- Incomplete parenthesis format

Programming Errors

Syntax Error : GRAMMATICAL ERROR

```
>>> print("Hello World)

SyntaxError: EOL while scanning string literal
>>>
>>> print(Programming is fun)

Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print(Programming is fun)
NameError: name 'Programming' is not defined
>>>
>>> printt("SE #7")

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    printt("SE #7")
NameError: name 'printt' is not defined
>>>
```

Programming Errors

Run-time Error: Input Error

- Conflict variable type
- Mathematical error
- Input error

```
>>> print("S" + "E" + "#7")
SE#7
>>>
>>> number = 7
>>> print("S" + "E" + "#" + number)

Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print("S" + "E" + "#" + number)
TypeError: cannot concatenate 'str' and 'int'
>>>
```

Programming Errors

Logic Error: Unexpected outcome

```
>>> PI = 3.14
>>> r = 2
>>> area = (PI * r) ** 2
>>> area
39.4384
```

```
>>> PI = 3.14
>>> r = 2
>>> area = PI * (r ** 2)
>>> area
12.56
>>> |
```

Introduction to Object-Oriented Programming:
Classes

Classes

The class statement

```
class Account:
   def __init__(self, initial): self.balance =
      initial
   def deposit(self, amt):
      self.balance = self.balance + amt def
   withdraw(self,amt):
      self.balance = self.balance - amt def
   getbalance(self):
      return self.balance
```

Using a class

```
a = Account(1000.00) a.deposit(550.23)
a.deposit(100) a.withdraw(50)
print a.getbalance()
```

Modules and Imports

Modules

Large programs can be broken into modules

```
# numbers.py
def divide(a,b):
   q = a/b
   r = a - q*b return q,r

def gcd(x,y): g = y
   while x > 0: g = x
      x = y % x y = g
   return g
```

The import statement

```
import numbers
x,y = numbers.divide(42,5)
n = numbers.gcd(7291823, 5683)
```

import creates a namespace and executes a file.

Reference

The University of Texas at Austin (cs.utexas.edu)

Advanced Python Programming, The University of Chicago (cs.uchicago.edu)

Data Structures and Algorithms, King Mongkut's Institute of Technology Ladkrabang (se.kmitl.ac.th)

Hands-On Exercise 1:
Coding Fundamentals 1

File Handling

Files

- The open() function

```
f = open("foo","w")                    # Open a file for writing
g = open("bar","r")                    # Open a file for reading
```

- Reading data

```
data = g.read()                        # Read all data
line = g.readline()                    # Read a single line
lines = g.readlines()                  # Read data as a list of lines
```

- Writing data

```
f.write("Hello World")
```

File Objects

```
open(filename [,mode])
```

- Opens a file and returns a file object
- By default, opens a file for reading
- Modes:

```
"r"                 Open for reading
"w"                 Open for writing (truncating to zero length)
"a"                 Open for append
"r+"                Open for read/write (updates)
"w+"                Open for read/write (with truncation to zero length)
```

- For example,

```
open("file.txt", "w")          # Open a file file.txt for writing
```

File Objects

File Methods

```
f.read([n])                    # Read at most n bytes
f.readline([n])                # Read a line of input with max length of n
f.readlines()                  # Read all input and return a list of lines
f.write(s)                     # Write string s
f.writelines(ls)               # Write a list of strings
f.close()                      # Close a file
f.tell()                       # Return current file pointer
f.seek(offset [,where])        # Seek to a new position
                          # where = 0: Relative to start
                          # where = 1: Relative to current
                          # where = 2: Relative to end
f.isatty()                     # Return 1 if interactive terminal
f.flush()                      # Flush output
f.truncate([size])             # Truncate file to at most size bytes
f.fileno()                     # Return integer file descriptor
```

File Objects

File Attributes

```
f.closed            # Set to 1 if file object has been closed
f.mode              # I/O mode of the file
f.name              # Name of file if created using open().
                # Otherwise, a string indicating the source
```

File and Path Manipulation

os.path

```
abspath(path)                    # Returns the absolute pathname of a path
basename(path)                   # Returns filename component of path
dirname(path)                    # Returns directory component of path
normpath(path)                   # Normalize a pathname
split(path)                      # Split path into (directory, file)
splitdrive(path)                 # Split path into (drive, pathname)
splitext(path)                   # Split path into (filename, suffix)
join(p1,p2,...)                  # Join pathname components
```

Examples

```
abspath("../foo")                # Returns "/home/beazley/blah/foo"
basename("/usr/bin/python")      # Returns "python"
dirname("/usr/bin/python")       # Returns "/usr/bin"
normpath("/usr/./bin/python")    # Returns "/usr/bin/python"
split("/usr/bin/python")         # Returns ("/usr/bin","python")
splitext("index.html")           # Returns ("index",".html")
```

File Tests

os.path

```
exists(path)          # Test for existence
isabs(path)           # Return 1 if path is an absolute pathname
isfile(path)          # Return 1 if path is a regular file
isdir(path)           # Return 1 if path is a directory
islink(path)          # Return 1 if path is a symlink
ismount(path)         # Return 1 if path is a mountpoint
```

Globbing

- Returns filenames in a directory that match a pattern

```
import glob
a = glob.glob("*.html")
b = glob.glob("image[0-5]*.gif")
```

- Pattern matching is performed using rules of Unix shell.

Hands-On Exercise 2:
Coding Fundamentals 2