LGT3109
Introduction to Coding for Business with Python
(week 4)

Xiaoyu Wang

Dept. of LMS, The HK PolyU

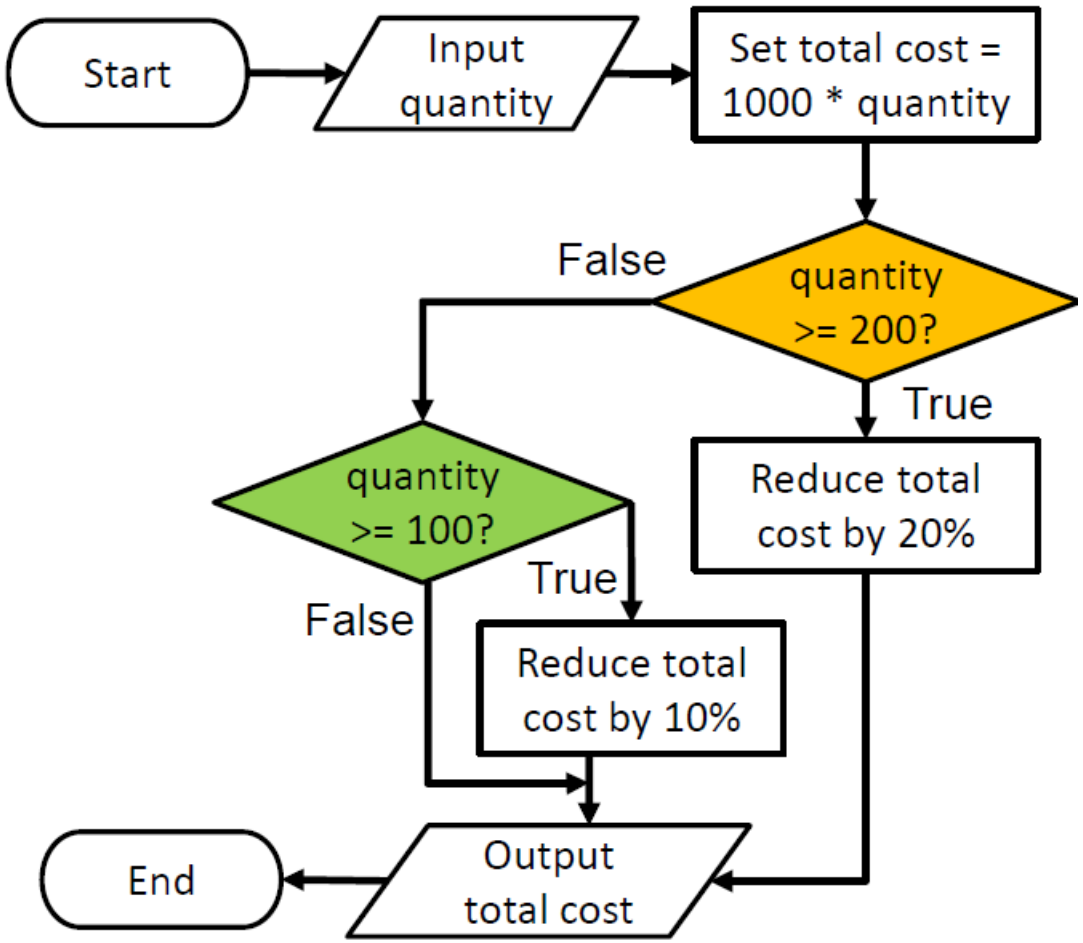
Summary of Week 3

- if condition: statements
- Boolean Expression
  - Comparison operators: == <= >= > < !=
  - Logic operators: and, or, not
- Indentation: indicate start/end block
- One-way Decisions
- Two-way decisions
  - if: statements else: statements
- Multi way decisions
  - If: statements elif : statements else: statements
- Nested Decisions: better to avoid.
- Problem to Flowchart to Code.

Summary of Week 3

- One-way/Two-way decisions
- Multiway decisions
- Nested decisions

```
#input:
quantity = int(input('What is the shipping quantity? '))
#process:
total_cost = 1000.0 * quantity
#conditional steps:
if quantity >= 200:
    total_cost = total_cost * (1.0 - 0.2)
elif quantity >= 100:
    total_cost = total_cost * (1.0 - 0.1)
#output:
print('The total cost is ' + str(total_cost) + '.')
```



Functions

- Basic Concepts

- Built-in Functions
  - Examples
  - Package/Module

- User-defined functions
  - Basics
  - Flow
  - Components
  - Positional/Keyword Arg
  - Local/Global variables
  - Comments

Functions-Foxconn Case

- A carrier quotes a shipping rate (USD/container) and discount:

➢Shipping rate = 1000 USD/container

➢If containers > threshold (100,200), discount (10%,20%) is applied.

```
#input:
quantity = int(input('What is the shipping quantity? '))
#process:
total_cost = 1000.0 * quantity
#conditional steps:
if quantity < 200 and quantity >= 100:
    total_cost = total_cost * (1.0 - 0.1)
if quantity >= 200:
    total_cost = total_cost * (1.0 - 0.2)
#output:
print('The total cost is ' + str(total_cost) + '.')
```

- Question: how to compute the cost for two different quantities?

Functions-Foxconn Case

- Question: how to compute the cost for two different quantities?

- Duplicate the code
  - ➢Lengthy
  - ➢Hard to maintain

```
#input:
quantity = int(input('What is the shipping quantity? '))
#process:
total_cost = 1000.0 * quantity
#conditional steps:
if quantity < 200 and quantity >= 100:
    total_cost = total_cost * (1.0 - 0.1)
if quantity >= 200:
    total_cost = total_cost * (1.0 - 0.2)
#output:
print('The total cost for quantity ', quantity, 'is ', str(total_cost) + '.')


#input:
quantity = int(input('What is the shipping quantity? '))
#process:
total_cost = 1000.0 * quantity
#conditional steps:
if quantity < 200 and quantity >= 100:
    total_cost = total_cost * (1.0 - 0.1)
if quantity >= 200:
    total_cost = total_cost * (1.0 - 0.2)
#output:
print('The total cost for quantity ', quantity, 'is ', str(total_cost) + '.')
```

Functions-Foxconn Case

- How to reuse the code?

➢Functions

➢Shorten the code

➢Easy to understand, edit, test, debug, maintain, …

```
#function to prompt a shipping quantity and compute the total cost:
def process():
    quantity = int(input('What is the shipping quantity? '))
    total_cost = 1000.0 * quantity
    if quantity < 200 and quantity >= 100:
        total_cost = total_cost * (1.0 - 0.1)
    if quantity >= 200:
        total_cost = total_cost * (1.0 - 0.2)
    print('The total cost for quantity ', quantity, 'is ', str(total_cost) + '.')


#for the 1st input quantity:
process()
#for the 2nd input quantity:
process()
```

Define Function

Call Function

Call Function

Functions

- Basic Concepts

- Built-in Functions
  - ➢Examples
  - ➢Package/Module

- User-defined functions
  - ➢Basics
  - ➢Flow
  - ➢Components
  - ➢Positional/Keyword Arg
  - ➢Local/Global variables
  - ➢Comments

Basic Concepts

- A function is a named sequence of statements that performs a computation.
- There are two kinds of functions in Python.
- Built-in functions: print(), input(), type(), float(), int(), max(), min().
- User-defined Functions: new functions that we define and use.
- We treat the built-in function names as “new” reserved words.

Basic Concepts-Two Type of Functions

- Built-in Functions
- For example: input(), print()

```
#input:
quantity = int(input('What is the shipping quantity? '))
#process:
total_cost = 1000.0 * quantity
#conditional steps:
if quantity < 200 and quantity >= 100:
    total_cost = total_cost * (1.0 - 0.1)
if quantity >= 200:
    total_cost = total_cost * (1.0 - 0.2)
#output:
print('The total cost is ' + str(total_cost) + '.')
```

Basic Concepts-Two Type of Functions

- User-defined Functions

Program without a user-defined function:

```
print('=========')
print('Hello', 'Alice')
print('=========')

print('|         |')

print('=========')
print('Hello', 'Bob')
print('=========')
```

Program with a user-defined function:

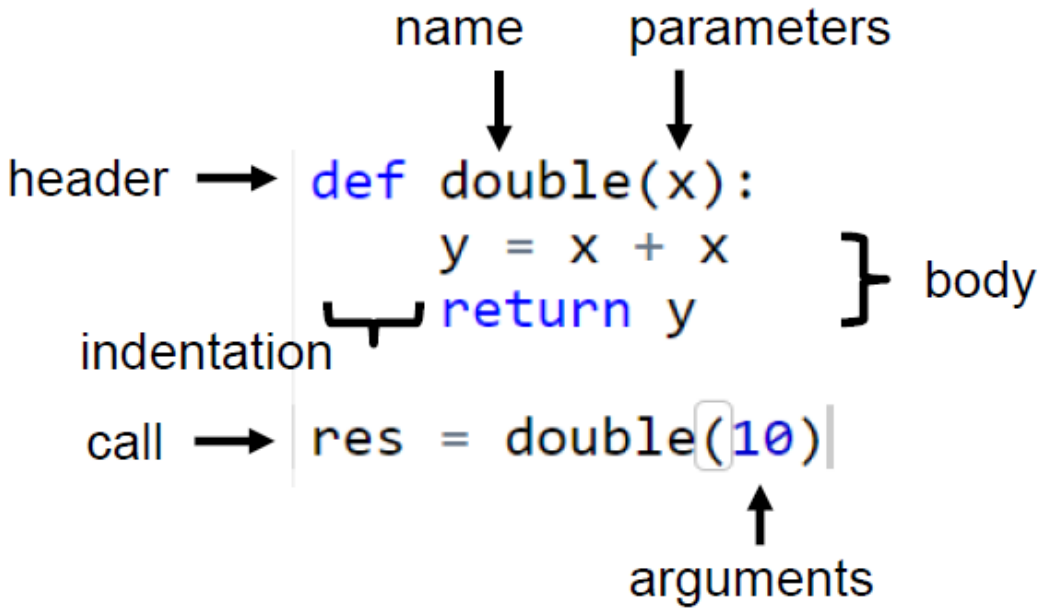
```
def greet(name):
    print('=========')
    print('Hello', name)
    print('=========')

greet('Alice')
print('|         |')
greet('Bob')
```

Output:

```
=========
Hello, Alice
=========
|         |
=========
Hello, Bob
=========
```

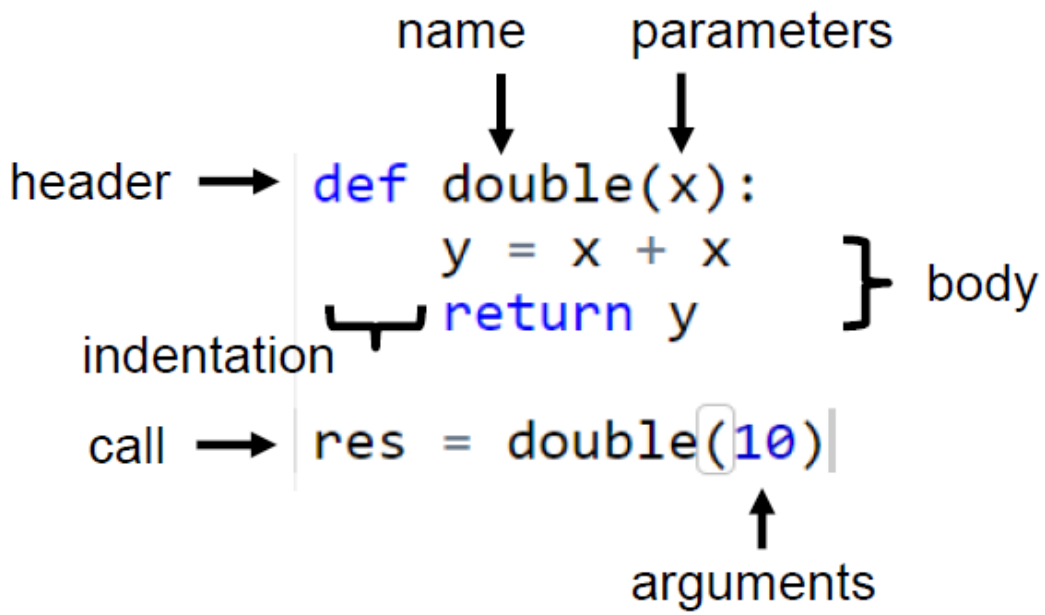
Basic Concepts-Coding

- In Python, a function takes argument(s) as input parameters, compute, and returns result(s).
- We define a function using the "def:" in its header.
- We call/invoke the function by using the function name and arguments in an expression.



Basic Concepts-Example

- Header, Name, Parameter, Body, Return
- Call, arguments




```
def process():
    #input
    quantity = int(input('What is the shipping quantity? '))
    #process
    total_cost= 1000*quantity

    #conditional steps
    if quantity <200 and quantity >= 100:
        total_cost=total_cost*(1-0.1)

    if quantity >=200:
        total_cost = total_cost*(1-0.2)

    #output
    print('The total cost is ' +str(total_cost)+'.')
```

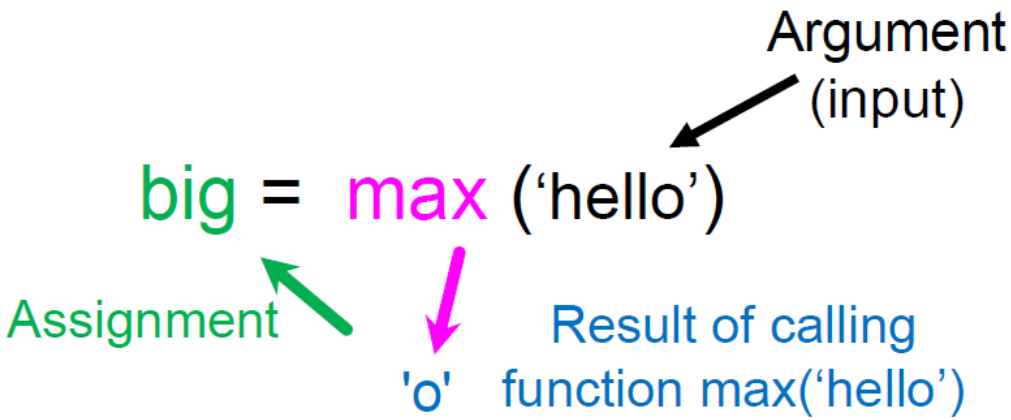
Functions

- Basic Concepts

- Built-in Functions
  - Examples
  - Package/Module

- User-defined functions
  - Basics
  - Flow
  - Components
  - Positional/Keyword Arg
  - Local/Global variables
  - Comments

Built-in Functions-max Functions

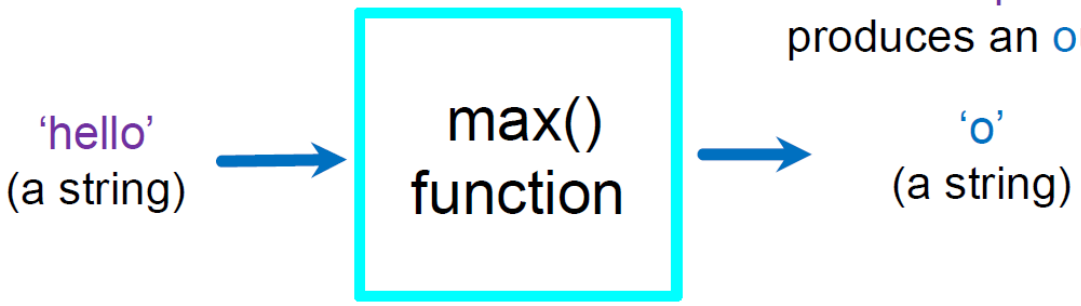



```
>>> big = max('hello')
>>> print(big)
o
>>> tiny = min('hello')
>>> print(tiny)
e
>>> num = len('hello')
>>> print(num)
>>> 5
```

Built-in Functions-max Functions

```
>>> big = max('hello')
>>> print(big)
o
```

A function is some stored code that we use. A function takes some input and produces an output.



Guido (the creator of Python) wrote this code

Built-in Functions-max Functions

```
>>> big = max('hello')
>>> print(big)
o
```

A function is some stored code that we use. A function takes some input and produces an output.

'hello'
(a string)

```
def max(inp):
    blah
    blah
    for x in inp:
      blah
      blah
```

'o'
(a string)

Guido (the creator of Python) wrote this code

Built-in Functions-float/int

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float.
- You can control this with the built-in functions int() and float(), to convert values or variables to integers and floats.

```
>>> x = 3
>>> print(type(x+4.0))
<class 'float'>

>>> print(float(99) / 100)
0.99
>>> i = 42
>>> type(i)
<class 'int'>

>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class 'float'>
```

Built-in Functions-float/int

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

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int'
```

```
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
```

Functions

- Basic Concepts

- Built-in Functions
  - Examples
  - Package/Module

- User-defined functions
  - Basics
  - Flow
  - Components
  - Positional/Keyword Arg
  - Local/Global variables
  - Comments

Built-in Functions-Math Functions

- math.pi: value of π
- math.sqrt(2): $\sqrt{2}$
- All these functions are from a module math.
- Module math needs to be imported before its use.

```
>>> import math
>>> print(math.pi)
3.141592653589793
>>> print(math.sqrt(2))
1.4142135623730951
>>>
```

Built-in Functions-Math Functions

- Module math provides most of the familiar mathematical functions.

- Before using the module, we need to import it:

➢>>> import math

- The module object contains the functions and variables defined in the module, which can now be used in your code.

Built-in Functions-Math Functions

- >>> import math
- To access the functions, you need to use dot notation.

➢specify the name of the module and the name of the function, separated by a dot.

- >>> print(math.pi)
- >>> 3.141592653589793
- >>> print(math.sqrt(2))
- >>> 1.4142135623730951

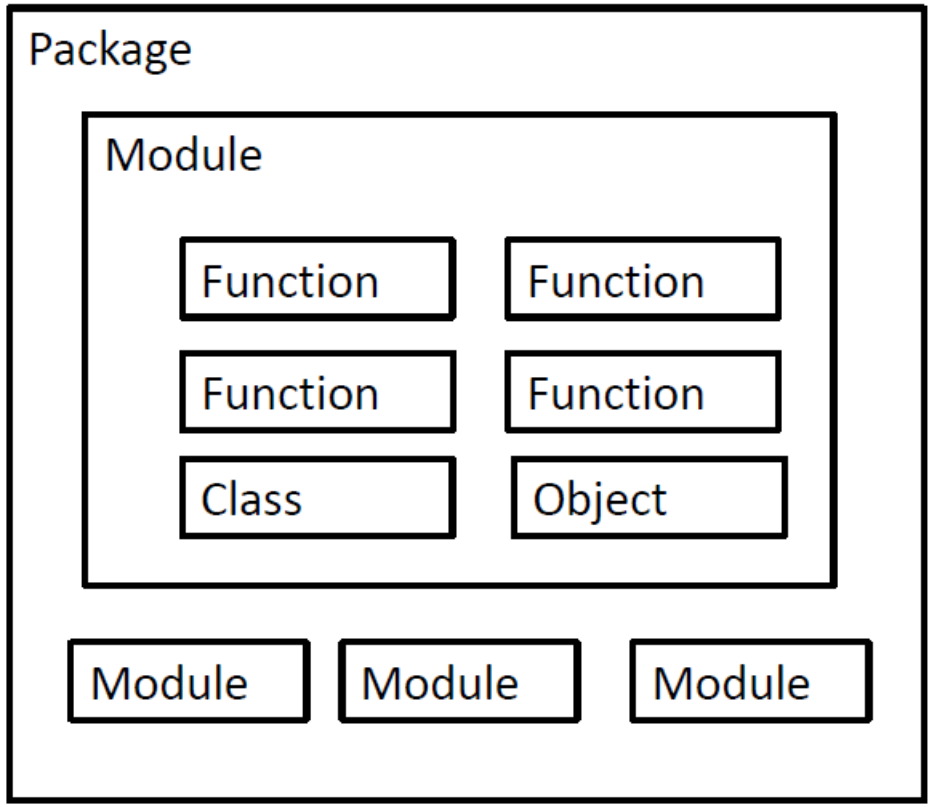
Built-in Functions-Package/Module

- Python codes can be reused through modules and packages.
- A package consists of related modules.

➢To install package: pip install scipy

- A module consists of related functions, classes, objects, etc.

➢To import modules: import math



Built-in Functions-Package/Module

- What built in functions used in this code?

➢input()

➢int()

➢print()

➢str()

- How to know their usage?

```
def process():
    #input
    quantity = int(input('What is the shipping quantity? '))
    #process
    total_cost= 1000*quantity

    #conditional steps
    if quantity <200 and quantity >= 100:
        total_cost=total_cost*(1-0.1)

    if quantity >=200:
        total_cost = total_cost*(1-0.2)

    #output
    print('The total cost is ' +str(total_cost)+'.')
```

Built-in Functions-Package/Module

- In Shell, use dir(__builtins __) to list all built-in functions. (__ not _)

➢ [… 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray ', 'bytes', 'callable', chr ', classmethod ', 'compile', 'print'…]

- Use help(f) to show the usage of function f

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

- Parameter **sep** has a default value ' '
- Parameter **end** has a default value '\n' (new line)

Built-in Functions-Package/Module

- How about installed modules and functions?
- import module
- Use dir(module) to list all functions in the package.
- Use help(module.function) to view the usage of the function in the package.

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh'
, 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign',
 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1', '
fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', '
inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma'
, 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'po
w', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
 'trunc', 'ulp']
>>>
```

```
>>> help(math.log)
Help on built-in function log in module math:

log(...)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base not specified, returns the natural logarithm (base e) of x.
```

Built-in Functions-Package/Module

- We sometimes need to pass more than one parameter to a function.
- Positioned arguments are used to identify arguments by their positions in the function call.
- Keyword arguments are used to set arguments to values other than their default values.

A positional argument is a name that is not followed by an equal sign (=) and default value.

A keyword argument is followed by an equal sign and an expression that gives its default value.

```
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

print (10, end='#'): 10 for value, and end = '#'

```
print (10, end='#')
print (20, end='#')
print (30)
```

```
10#20#30
```

Summary

- What’s function?
  - ➢To store and reuse codes
  - ➢define, header, name, arguments, body, return, call, parameters
- Built-in functions
  - ➢Import module before using its functions/variables
  - ➢Usage of dir() and help()
  - ➢Passing arguments by positions and keywords

Functions

- Basic Concepts

- Built-in Functions
  - Examples
  - Package/Module

- User-defined functions
  - Basics
  - Flow
  - Components
  - Positional/Keyword Arg
  - Local/Global variables
  - Comments

User-defined Functions

- We create a new function using the def keyword followed by optional parameters (arguments) in parentheses in the header.
- The body of the function is indented.
- This defines the function but does not execute the body of the function.

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

User-defined Functions-Definitions/Uses

- Once we have defined a function, we can call (or invoke) it as many times as we like.
- This is the store and reuse pattern.
- Function definition must be executed before it is called.

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

```
print_lyrics()
print_lyrics()
```

```
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

User-defined Functions-Definitions/Uses

- Function definition must be executed before it is called.

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')

repeat_lyrics()
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

```
Traceback (most recent call last):
  File
"/tmp/sessions/8ae499cceffb7895/main.py",
line 5, in <module>
    repeat_lyrics()
NameError: name 'repeat_lyrics' is not
defined
```

Functions

- Basic Concepts

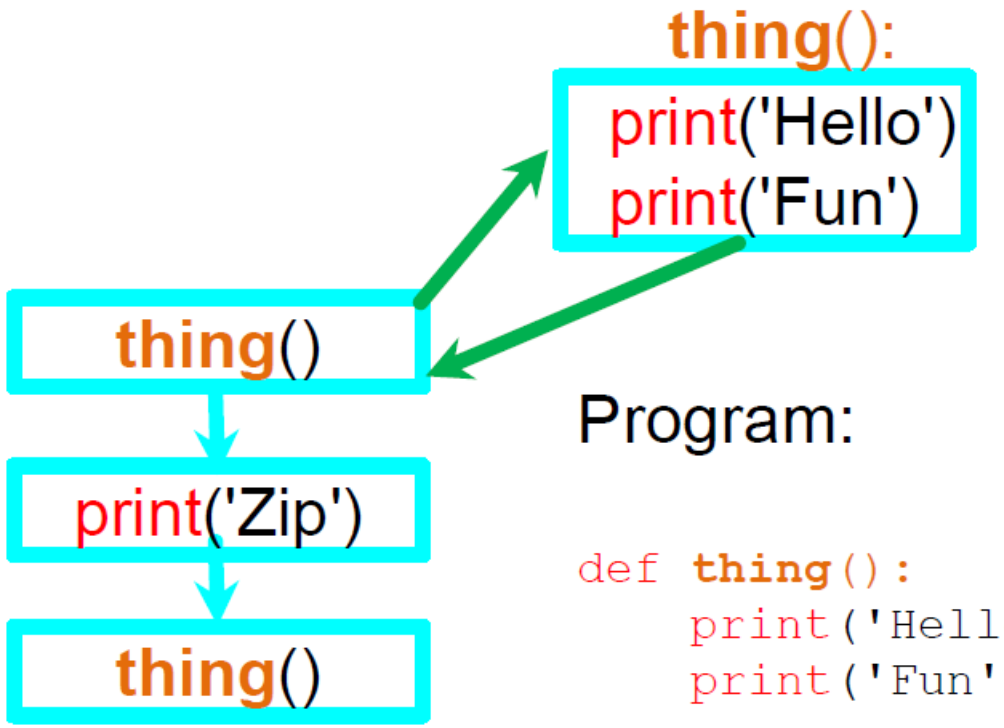
- Built-in Functions
  - ➢Examples
  - ➢Package/Module

- User-defined functions
  - ➢Basics
  - ➢Flow
  - ➢Components
  - ➢Positional/Keyword Arg
  - ➢Local/Global variables
  - ➢Comments

User-defined Functions-Flow of Execution

- Statements are executed one by one, from top to bottom.
- Statements inside the function are not executed until the function is called.
- A function call is like a detour in the flow of execution.

➢The flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

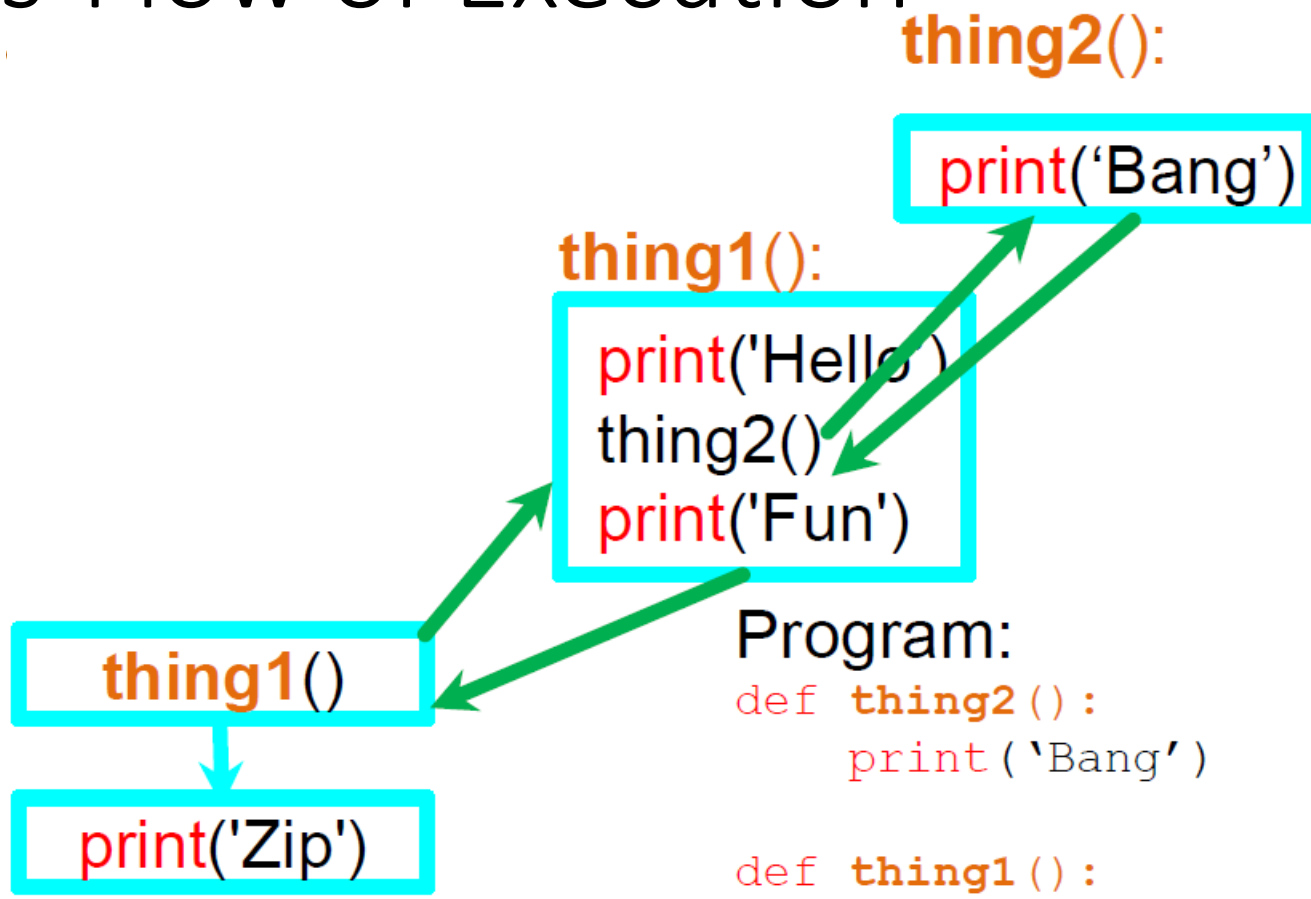


User-defined Functions-Flow of Execution

- One function can call another.

➢While executing that new function, the program might have to execute yet another function!

➢Each time a function completes, the program picks up where it left off in the function that called it.



Functions

- Basic Concepts

- Built-in Functions
  - ➢Examples
  - ➢Package/Module

- User-defined functions
  - ➢Basics
  - ➢Flow
  - ➢Components
  - ➢Positional/Keyword Arg
  - ➢Local/Global variables
  - ➢Comments

User-defined Functions-Argument

- An argument is a value we pass into the function as its input when we call the function.
- We use arguments so that we can direct the function to do different kinds of work when we call it at different times.
- We put the arguments in parentheses after the name of the function.

```
>>> def greet(lang):
...     if lang == 'es':
...         print('Hola')
...     elif lang == 'fr':
...         print('Bonjour')
...     else:
...         print('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```

User-defined Functions-Parameter

- A parameter is a variable which we use in the function definition to access arguments.
- –It is a handler that allows the code in the function to access the arguments for a particular function invocation.

```
>>> def greet(lang):
...     if lang == 'es':
...         print('Hola')
...     elif lang == 'fr':
...         print('Bonjour')
...     else:
...         print('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```

User-defined Functions-Return

- Often a function will take its arguments, do some computation and return a value.
- The return keyword is used for this.

```
def greet():
    return "Hello"

print(greet(), "Glenn")
res = greet ()
print(res, "Sally")
```

```
Hello Glenn
Hello Sally
```

User-defined Functions-Return

- A “fruitful” function is one that produces a result (or return value).
- The return statement ends the function execution and sends back the result of the function.

```
>>> def greet(lang):
...     if lang == 'es':
...         print('Hola')
...     elif lang == 'fr':
...         print('Bonjour')
...     else:
...         print('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```

User-defined Functions-Return

- When a function does not return a value, we call it a “void” function.
- Functions that return values are “fruitful” functions.
- Void functions are “not fruitful”.

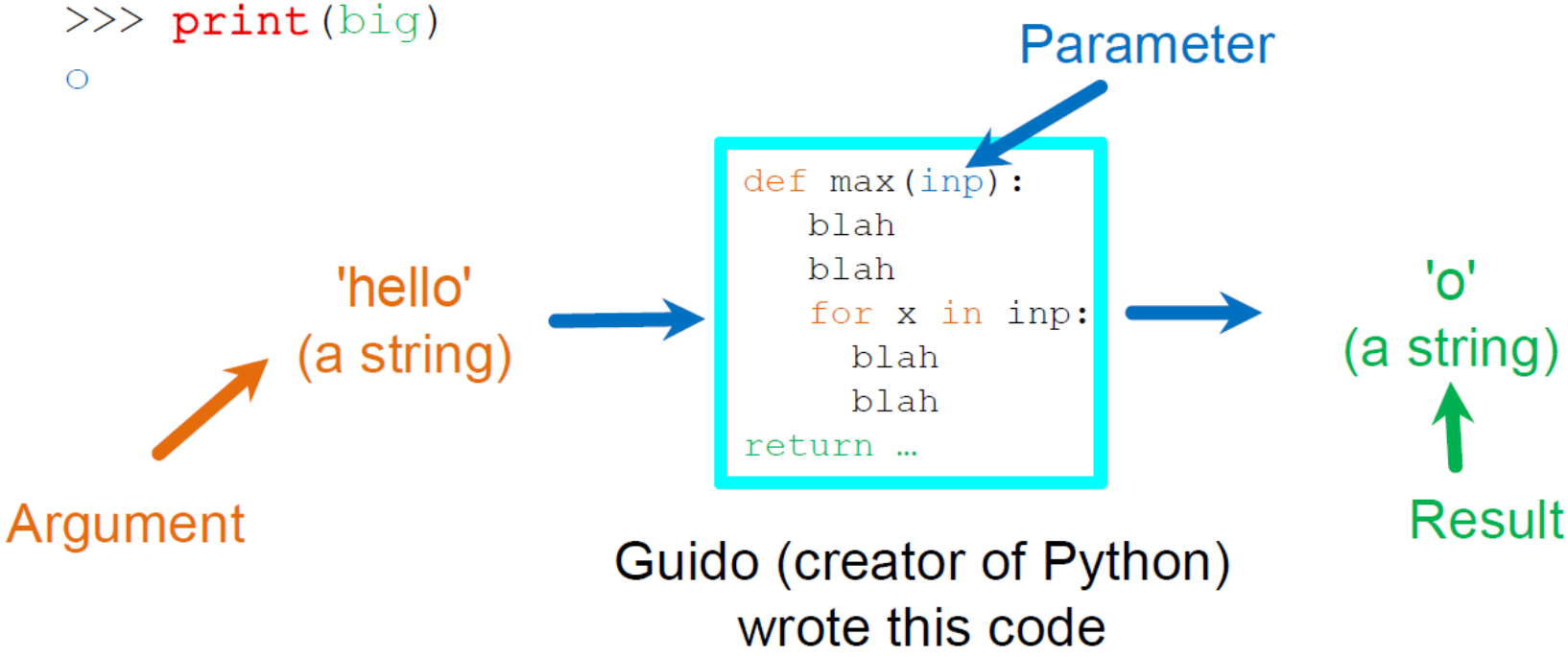
```
#a void function:
def print_line():
    print('==========')

print_line()
print('LGT3109')
print_line()
```

```
#another void function:
def print_line():
    print('==========')
    return

print_line()
print('LGT3109')
print_line()
```

User-defined Functions-Revisit

Arguments, Parameters, and Results

```
>>> big = max('hello')
>>> print(big)
o
```



User-defined Functions-Multiple Parameters

- We can define more than one parameter in the function definition.
- We simply add more arguments when we call the function.
- We match the number and order of arguments and parameters.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)

8
```

3 in arguments matches variable a in parameters
5 in arguments matches variable b in parameters

Functions

- Basic Concepts

- Built-in Functions
  - ➢Examples
  - ➢Package/Module

- User-defined functions
  - ➢Basics
  - ➢Flow
  - ➢Components
  - ➢Positional/Keyword Arg
  - ➢Local/Global variables
  - ➢Comments

Comments-Positional/Keyword Arguments

- Positional Arguments
- Match each argument with a parameter in function definition based on the order.

```
def combine_name(first_name, last_name):
    res = first_name+ ' ' + last_name
    return res
full_name= combine_name('Xiaoyu', 'Wang')
print(full_name)
```

Comments-Positional/Keyword Arguments

- Keyword Arguments
- Pass name-value pairs to a function.
- Directly associate the name and the value within the argument of the function.
- Keyword arguments must be specified after positional arguments!

```
def combine_name(first_name, last_name):
    res = first_name+ ' ' + last_name
    return res
full_name= combine_name(first_name='Xiaoyu', last_name='Wang')
print(full_name)
```

Comments-Positional/Keyword Arguments

- Default Values
- In a function, you can define a default value for each parameter.
- If an argument's value for a parameter is provided in the function call, the value is used, otherwise, the default value is used.
- Default arguments must be specified after non-default arguments!

name_default.py - C:\Users\xiaoy\Desktop\name_default.py (3.11.4)

File Edit Format Run Options Window Help

```
def combine_name(first_name,last_name=''):
    res =first_name + ' ' + last_name
    return res

full_name= combine_name(first_name ='Xiaoyu', last_name ='Wang')
full_name= combine_name('Xiaoyu', last_name ='Wang')
#full_name= combine_name(first_name='Xiaoyu','Wang')
full_name= combine_name('Xiaoyu')
print(full_name)
```

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
```

```
8
```

Functions

- Basic Concepts

- Built-in Functions
  - ➢Examples
  - ➢Package/Module

- User-defined functions
  - ➢Basics
  - ➢Flow
  - ➢Components
  - ➢Positional/Keyword Arg
  - ➢Local/Global variables
  - ➢Comments

Comments-Local Variables

- Local variables: variables declared (by an assignment statement) inside a function.

➢Can be accessed only inside the function.

➢The function is the (local) scope of such a variable.

```
# define and use local variables:
def foo():
    y = "local"
    print(y)

foo()
```

```
local
```

```
def foo():
    y = "local"


foo()
print(y)
```

```
----------------------------------------
NameError                                 Tr
<ipython-input-39-60986766f2cf> in <module>
      4
      5 foo()
----> 6 print(y)

NameError: name 'y' is not defined
```

```
>>> def print_rate():
        rate = 5.0
        print("rate =", rate)


>>> print_rate()
rate = 5.0
>>> rate = 10.0
>>> print_rate()
rate = 5.0
```

Comments-Global Variables

- Global variables: variables declared (by an assignment statement) outside a function.
- Can be accessed inside or outside a function.
- Have a global scope.
- Need to be declared as 'global' if changing its value in a function.

```
x = "global"
def foo():
    print("x inside:", x)

foo()
print("x outside:", x)
```

```
x inside: global
x outside: global
```

```
x = "global"

def foo():
    x = x + x
    print(x)

foo()
```

```
UnboundLocalError: local variable 'x' referenced before assignment
```

```
x = "global"

def foo():
    global x
    x = x + x
    print(x)

foo()
```

```
globalglobal
```

Comments-Local/Global Variables

- For variables with the same name inside and outside of a function, Python will treat them as two separate variables.
- One available in the global scope (outside the function).
- One available in the local scope (inside the function).

```
# global and local variables have the same name:
x = 5

def foo():
    x = 10
    print("local x:", x)


foo()
print("global x:", x)
```

Functions

- Basic Concepts

- Built-in Functions
  - ➢Examples
  - ➢Package/Module

- User-defined functions
  - ➢Basics
  - ➢Flow
  - ➢Components
  - ➢Positional/Keyword Arg
  - ➢Local/Global variables
  - ➢Comments

Comments-Chunks

- Organize your code into “paragraphs” capture a complete thought and “name it”.
- Don’t repeat yourself: make it work once and then reuse it.
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions.

Comments-Chunks

```
#function to prompt a shipping quantity and compute the total cost:
def process():
    quantity = int(input('What is the shipping quantity? '))
    total_cost = 1000.0 * quantity
    if quantity < 200 and quantity >= 100:
        total_cost = total_cost * (1.0 - 0.1)
    if quantity >= 200:
        total_cost = total_cost * (1.0 - 0.2)
    print('The total cost for quantity ', quantity, 'is ', str(total_cost) + '.')


#for the 1st input quantity:
process()
#for the 2nd input quantity:
process()
```

Question: How to further improve the organization of the code?

Comments-Chunks

```
#Function to compute total cost:
def compute_total_cost(quantity):
    total_cost = 1000.0 * quantity
    if quantity < 200 and quantity >= 100:
        total_cost = total_cost * (1.0 - 0.1)
    if quantity >= 200:
        total_cost = total_cost * (1.0 - 0.2)
    return total_cost

def process():
    quantity = int(input('What is the shipping quantity? '))
    total_cost = compute_total_cost(quantity)
    print('The total cost for quantity ', quantity, 'is ', str(total_cost) + '.')


process()
process()
```

break it up into logical chunks:

Chunk (Paragraph) 1

Chunk 2

Comments-Debugging

- Make sure the code you are looking at is the code you are running.

➢If you're not sure, put something like print("hello") at the beginning of the program and run it again.

➢If you don't see hello, you're not running the right program!

```
print('hello code')
blah blah
Blah blah
Print('bye code')
```

```
def f():
    print('hello f’)
    blah blah
    Blah blah
    print('bye f')
```

Comments-Revisit

```
#Function to compute total cost:
def compute_total_cost(quantity):
    total_cost = 1000.0 * quantity
    if quantity < 200 and quantity >= 100:
        total_cost = total_cost * (1.0 - 0.1)
    if quantity >= 200:
        total_cost = total_cost * (1.0 - 0.2)
    return total_cost

def process():
    quantity = int(input('What is the shipping quantity? '))
    total_cost = compute_total_cost(quantity)
    print('The total cost for quantity ', quantity, 'is ', str(total_cost) + '.')


process()
process()
```

Acknowledgement

- Acknowledgements / Contributions

- Initial Development: Charles Severance, University of Michigan School of Information
- Further Development: Zhou Xu, Hong Kong Polytechnic University
- Continuous development: Xiaoyu Wang, Hong Kong Polytechnic University