Computer Programming I

User-Defined Functions

Review: creating our own functions

`def` needs three things to follow it on the same line:

1. The name we want to give to our function
   - Follow the same rules as variable names / identifiers
2. The names for the parameters between parentheses
   - If there is more than one parameter, separate them with commas
   - If there are no parameters, just type the parentheses
3. End the line with a colon `:`

After the `def` line comes the body of the function

Examples

```
def print_greeting(userName) :
    print('Hello' + userName)
    print('Welcome to the functions demo!')

def calculate_age(currentYear, birthYear):
    age = currentYear - birthYear
    return age

print_greeting('Jeff’)       #call to print_greeting
userAge = calculate_age(2020, 2000) #call to calculate_age
```

Parameters vs. Arguments

- The data that sent into a function is known as arguments
  - Functions use the arguments in calculations or other operations
- If you want a function to receive arguments when it is called, you must equip the function with one or more parameter variables
  - A parameter variable is often simply called a parameter
- A parameter is a special placeholder that will receive the value of an argument when a function is called

Examples

```
def print_greeting(userName) : #username is a parameter
    print('Hello’ + userName)
    print('Welcome to the functions demo!')

print_greeting('Jeff') #'Jeff' is an argument
print_greeting('Rose') # 'Rose' is an argument
name = 'Charlie'
print_greeting(name) #'Charlie' is an argument
-------------------------------------------------------
Hello Jeff
Welcome to the functions demo!
Hello Rose
Welcome to the functions demo!
Hello Charlie
Welcome to the functions demo!
```

Passing Arguments to Functions

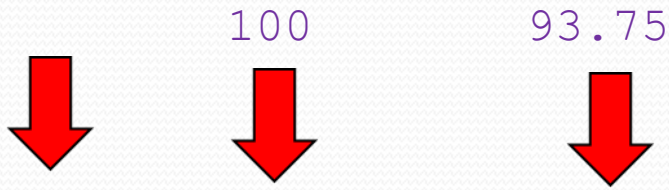
- Arguments and parameters do not have to have the same names
- Literals, variables and expressions can be used as arguments when calling a function.
- When calling a function:
  - The number of arguments in the function call must match the number of parameters in function definition
  - The first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.
  - The type of the arguments in the function call must compatible with what the function does

Passing Arguments to Functions

```
finalExam = 100
midtermExam = 75
labAverage = 88
grade = calculate_grade(midtermExam, finalExam, labAverage)
```


```
     def calculate_grade(midterm,      final,      labs):
```

```
labSum = 750
grade = calculate_grade(   95.5, finalExam, labSum / 8)
```




```
     def calculate_grade(midterm,    final,     labs):
```

Returning values from functions

- The `return` statement:
  - causes a value to be returned from the function
  - ends the execution of the function
- Only a single object can be returned from a function at any given time
- A function may have multiple `return` statements in its body
  - but only one of them will be executed in each function call

Example

```
def calculate_grade(midterm, final, labs) :
    return (midterm * 0.25 + final * 0.25 + labs * 0.50)

def assign_letter(grade) :
    if (grade >= 90) :
        return 'A'
    elif (grade >= 80) :
        return 'B'
    elif (grade  >= 70) :
        return 'C'
    elif (grades >= 60) :
        return 'D'
    else :
        return 'F'
```

Nested Function Calls

- You can call a function inside another function call
- We've done this before:

```
finalExam = float(input('Enter final exam grade: '))
print('Final grade:', calc_grade(mid, fin, labAvg))
```

- The inner function is evaluated first, and its returned value passed to the outer function call

None

- If you use **`return`** with no values, it will return "nothing"
- In Python "nothing" is referred to as **`None`**.
- Example:

```
def name_fun (a, b):
    print('Inside the function the value of a is:', a)
    print('Inside the function the value of b is:', b)
    return

a = 7
b = 9
print ('Before the function the value of a is:', a )
print ('Before the function the value of b is:', b )
print ('Return Value:', name_fun(a, b))
print ('After the function the value of a is:' , a )
print ('After the function the value of a is:' , b )
```

None

```
>>> %Run returnNone.py
Before the function the value of a is: 7
Before the function the value of b is: 9
Inside the function the value of a is: 7
Inside the function the value of b is: 9
Return Value: None
After the function the value of a is: 7
After the function the value of a is: 9
```

Returning containers

- What would you do if you need to '*return*' multiple values from a function?
  - return statements are limited to returning only one value
- A workaround is to package the multiple values you wish to return into a single container (such a tuple or a list) and to then return that container.

Example

```
def make_change(pennies) :
    centsPerQuarter = 25
    centsPerDime = 10
    centsPerNickel = 5
    quarters = pennies // centsPerQuarter
    pennies = pennies % centsPerQuarter
    dimes = pennies // centsPerDime
    pennies = pennies % centsPerDime
    nickels = pennies // centsPerNickel
    pennies = pennies % centsPerNickel

    return (quarters, dimes, nickels, pennies)

q, d, n, p = make_change(83)
```

Getting input

- Asking the user for input is a common problem.
- We can write a function to solve this problem.

```
def readInt(prompt) :
    valid = False
    while not valid :
        try :
            x = int(input(prompt))
            valid = True
        except ValueError as e:
            print ('Not an integer number. Try Again.')
    return x
```

Getting input

```
>>> value = readInt('Enter an integer: ')
Enter an integer: abc
Not an integer number. Try Again.
Enter an integer: x
Not an integer number. Try Again.
Enter an integer: 1.5
Not an integer number. Try Again.
Enter an integer: 5
```

Getting input

```
# calculate the average of 3 values entered by the user
total = 0
for i in range(1, 4):
    value = readInt('Quiz ' + str(i) + ': ')
    total = total + value
average = total / 3
print ('the average is:', average)
```

Getting input

```
>>> %Run returnNone.py
Quiz 1: 100
Quiz 2: abc
Not an integer number. Try Again.
Quiz 2: 90
Quiz 3: 1.4
Not an integer number. Try Again.
Quiz 3: 67
the average is: 85.66
```

Benefits of using Functions

- Simpler Code
  - A program's code tends to be simpler and easier to understand/read when it is broken down into functions
- Code Reuse
  - Functions reduce the duplication of code within a program.
  - A function can be called multiple times, as many times as needed
- Better Testing
  - Can test and debug each function individually and independently from other functions
- Faster Development
  - Functions can be written for commonly needed tasks, and those functions can be incorporated into each program that needs them
- Easier Facilitation of Teamwork / Modular development
  - Different programmers can be assigned the task of writing different functions

Documenting functions

- All functions should have internal documentation explaining their use.
  - This is done via comments, or docstrings
  - Provide enough information to effectively use the function.
  - Implementation details generally do not need to be given.
- Typical documentation often includes
  - the purpose of the function
  - a description of the parameters and return values
  - any other important information the user needs to know.
- A docstring is a string literal that begins and end with triple quotes
  - It can span multiple lines
  - `''' this is an example of a docstring'''`

Example

```
def get_int_in_range(low, high)
  ''' Purpose:
     prompts the user for an integer between low and high
     and returns a valid input.
     Parameters:
     low: an integer that represents the lower bound
     high: an integer that represents the upper bound
     return value:
     a number between the upper and lower bounds
     Usage example:
     n = get_int_in_range (5, 10)
  '''
  #body of the function goes here
```

Default arguments

- A programmer can assign a **default argument value** to a parameter in the function definition, allowing calls to the function to omit an argument for that parameter.
- Only parameters at the end of the parameter list can be omitted.
- If a parameter does not have a default value, then failing to provide an argument generates an error.

Example

```
def print_date(day, month, year, style = 0):
    if style == 0: # American
        print(month, '/', day, '/', year)
    elif style == 1: # European
        print(day, '/', month, '/', year)
    else:
        print('Invalid Style’)

print_date(12, 2, 2019, 0)
print_date(12, 2, 2019, 1)
print_date(12, 7, 2019)
```

The `split()` method

- `split(separator)` separates a string into a list of sub-strings, using `separator` to separate the strings.
  - It returns a list of strings.
- Example:

```
>>> phrase = "where there's hope, there's life"
>>> words = phrase.split() #separator is a blank space
>>> words
['where', "there's", 'hope,', "there's", 'life']
>>> site = 'www.python.org'
>>> parts = site.split('.') #separator is .
>>> parts
['www', 'python', 'org']
```

Creating our own version

```
def mySplit(line, separator = ' ' ) :
    pos = 0
    word = ''
    strList = []
    while pos < len (line) :
        if line [pos] == separator :
            strList.append (word)
            word = ''
        else :
            word += line[pos]
        pos += 1
    strList.append(word)
    return strList
```

Creating our own version

```
>>> text = 'That Sam-I-am That Sam-I-am! I do not like That Sam-I-am'
>>> words = text.split()
>>> words2 = mySplit(text)
>>> words
['That', 'Sam-I-am', 'That', 'Sam-I-am!', 'I', 'do', 'not', 'like',
'That', 'Sam-I-am']
>>> words2
['That', 'Sam-I-am', 'That', 'Sam-I-am!', 'I', 'do', 'not', 'like',
'That', 'Sam-I-am']
>>>
>>> words2 = mySplit(text, 'a')
>>> words2
['Th', 't S', 'm-I-', 'm Th', 't S', 'm-I-', 'm! I do not like Th', 't
S', 'm-I-', 'm']
>>>
```

Scope

- Scope is the part of the program where a variable or function is visible.
  - Visible → where the variable/function can be used
- The scope of a variable created inside a function from the variable's assignment until the end of the function
  - Local variable
- A variable defined outside of a function is called a global variable.
  - A global variable is visible from assignment to the end of the file that contains the program.
  - A global variable can be accessed by other functions.

Global variables

- If you wish to change the value of a global variable inside a function, you must use the keyword **`global`**.
- Example:

```
employee_name = 'N/A' #global variable

def get_name():
    global employee_name
    name = input('Enter employee name:')
    employee_name = name

get_name()
print('Employee name:', employee_name)
```

Global variables

- Assignment of global variables in functions should be used sparingly.
- If a local variable has the same name as a global variable, then the name refers to the local item
  - so the global variable is inaccessible inside the function
- If a function updates the global variable, the function may have side effects that are hard for a programmer to recognize.
- Use of global variables is typically limited to defining constants that are independent of any function.

Functions and scope

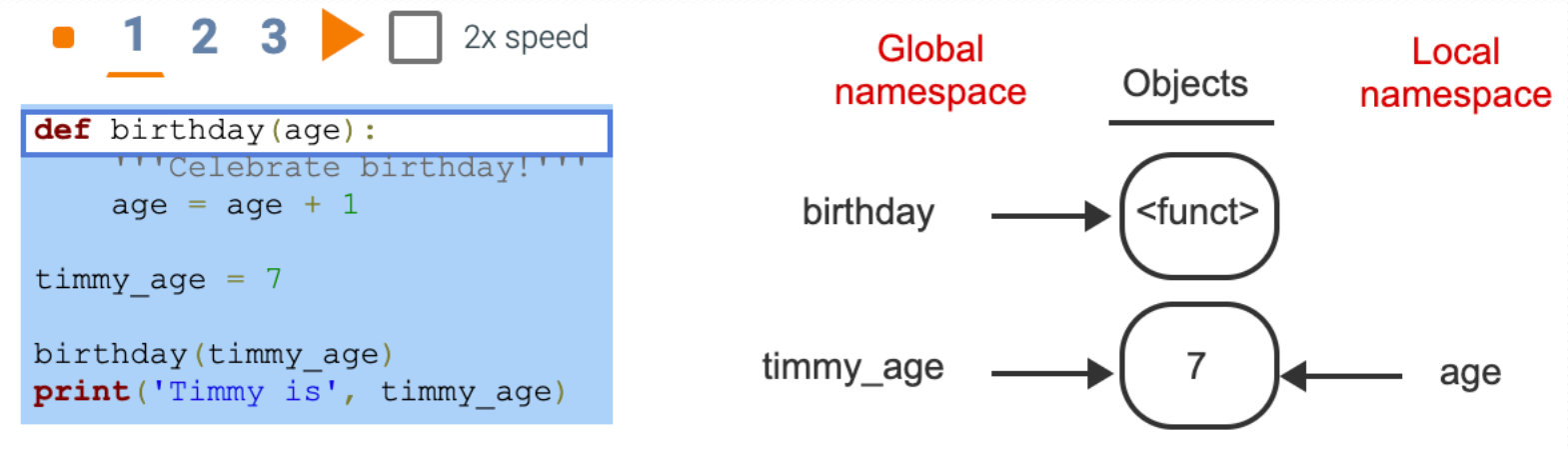
- A function’s also have scope extends from the function's definition to the end of the file.
- You must define a function before you can call it.
  - An attempt to call a function before a function has been defined results in an error.

Scope resolution

- Namespace: used to track all objects in a program.
  - Maps names to objects
  - Used to make scope work
  - Each type of scope has its own namespace
- 3 types of scope:
  - Built-in: for all built-in names in Python (e.g. `int`, `print`)
  - Global: global defined names (global variables, functions)
  - Local: within the currently executing function
- Scope resolution: process of searching for a name in the namespaces
- Multiple variables may have the same name but different values as long they have different scope

Function Arguments

- Arguments to functions are passed by object reference, a concept known in Python as pass-by-assignment.
- When a function is called, new local variables are created in the function's local namespace by binding the names in the parameter list to the passed arguments.
  - See animation 12.13.1



Function Arguments

When a function modifies a parameter:

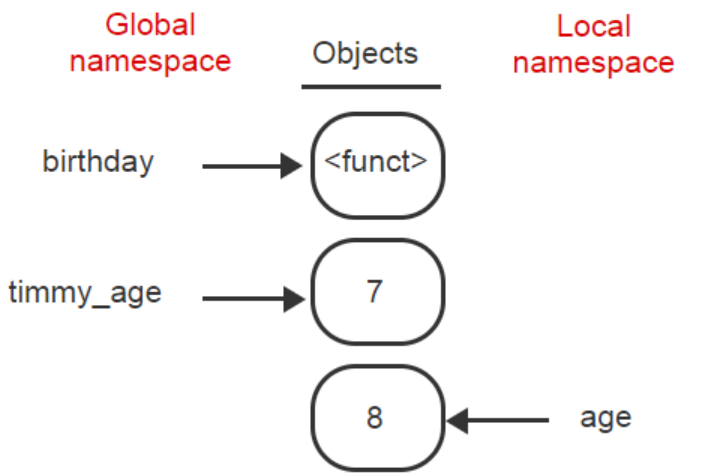
- If the object is immutable then the modification is limited to inside the function.
  - Any modification results in the creation of a new object in the function's local scope, leaving the original argument object unchanged

1 2 3 2x speed

```
def birthday(age):
    '''Celebrate birthday!'''
    age = age + 1

timmy_age = 7

birthday(timmy_age)
print('Timmy is', timmy_age)
```



Function Arguments

When a function modifies a parameter:

- If the object is mutable, then the modification of the object in the function can be seen outside the scope of the function.
  - One way to avoid unwanted changes is to pass a copy of the object as the argument instead