Data Processing and Analysis in Python
Lecture 7
Functions

UNIVERSITY OF MARYLAND

ROBERT H. SMITH SCHOOL OF BUSINESS

DR. ADAM LEE

What Functions Are and How They Work

- A **function** packages an algorithm in a chunk of code that you can **call** by **name**
- A function can be called from anywhere in a program's code, including code within other functions
- A function can **receive data from its caller** via **arguments**
- When a function is called, any expression supplied as arguments are first evaluated
- A function may have one or more **return** statements

Types of Functions

- **Built-in** functions:
  - Are part of Python or provided through libraries/modules
  - Examples: print(), input(), ascii(), chr(), abs(), round(), etc.
  - Class functions: int(), float(), str(), list(), tuple(), set(), dict(), etc.
- **User-defined** functions:
  - Defined by a programmer and can be used by programmers
  - A user-defined function is executed only when it is called/invoked
- **Anonymous** functions:
  - a.k.a. lambda function
  - Is not declared with the def keyword

Defining Simple Functions

- Defining our own functions allows us to organize our code in existing scripts more effectively
- Syntax of a function definition:
  def <function-name>(<parameter−1>, ..., <parameter−*n*>):
    ["""<doc-string>"""]
    <body>
  - Example:

```
def square(x):
    """Returns the square of x."""
    return x * x
```

  - Doc-string contains information about what the function does
  - to display, enter **help(square)**

Parameters and Arguments

- A **parameter** is the name used in the function definition for an **argument** that is passed to the function when it is called
- The number and positions of arguments of a function call should match the number and positions of the parameters
- Some function expects no arguments
  - It was defined with no parameters
- Python has four types of arguments:
  - Required positional arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments

Required Arguments

- a.k.a. required positional arguments
- Arguments passed to a function in correct positional order
- The number of arguments in the function call must match exactly with the parameters in the function definition

```
def avg(first, second):
    """Returns the average of two numbers - first and
second."""
    return (first + second) / 2
>>> avg(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    avg(1)
TypeError: avg() missing 1 required positional argument:
'second'
```

Keyword Arguments

- a.k.a. named arguments
- Identifying keyword arguments in a function invocation, the caller identifies the arguments by parameter name
- Allows to skip arguments or place them out of order
- Python matches the keyword names with the parameters and their values

```
def avg(first, second):
    """Returns the average of two numbers -
first and second."""
    return (first + second) / 2
>>> avg(second=2, first=1)
1.5
```

Default Arguments

- An argument that assumes a default value if not provided in the function call for the argument

```
def avg(first, second = 0):
    """Returns the average of two numbers -
first and second."""
    return (first + second) / 2
>>> avg(1, 2)
1.5
>>> avg(1)
0.5
```

Variable-Length Arguments

- You may need to process a function for more arguments than you specified while defining the function or it may be unclear how many arguments will be provided
- Use * before the parameter name that holds the values of all other variable arguments in the definition
- Only one variable-length argument is permitted per function definition
- This parameter remains empty if no additional arguments are specified during the function call

```
def avg(*number):
    """Returns the average of the given numbers."""
    return sum(number) / len(number)
```

The Return Statement

- The **return** keyword ends the function execution and sends back the function result
- Place a **return** statement at each exit point of a function when function should explicitly return a value
- Syntax:
  return <expression>
- If a function contains no return statement, Python transfers control to the caller after the last statement in the function's body is executed
  - The special value **None** is automatically returned

Functions as Abstraction Mechanisms

- An **abstraction** hides detail
  - Allows a person to view many things as just one thing
- Functions serve as abstraction mechanisms is by hiding complicated details
  - The idea of summing a range of numbers is simple; the code for computing a summation is not
- A function call expresses the idea of a process to the programmer
  - Without forcing to wade through the complex code that realizes that idea

Functions Eliminate Redundancy

- Functions serve as abstraction mechanisms by eliminating redundant, or repetitious, code

```
def summation(lower, upper):
    result = 0
    while lower <= upper:
        result += lower
        lower += 1
    return result
>>> summation(1, 4) # Summation of numbers 1..4
10
>>> summation(50, 100) # Summation of numbers
50..100
3825
```

UNIVERSITY OF MARYLAND

Functions Support the Division of Labor

- In a well-organized system, each part does its own job in collaborating to achieve a common goal
- In a computer program, functions can enforce a division of labor
  - Each function should perform a single coherent task
  - Example: Computing a summation
- Each of the tasks required by a system can be assigned to a function
  - Including the tasks of managing or coordinating the use of other functions

Problem Solving with Top-Down Design

- **Top-down design** starts with a global view of the entire problem and breaks the problem into smaller, more manageable sub-problems
  - Process known as **problem decomposition**
- As each sub-problem is isolated, its solution is assigned to a function
- As functions are developed to solve sub-problems, solution to overall problem is gradually filled out
  - Process is also called **stepwise refinement**

The Design of the Text-Analysis Program

- **Structure chart** – a diagram that shows the relationship among a program's functions and the passage of data between them
  - Each box in the structure is labeled with a function name
  - The main function at the top is where the design begins
  - Lines connecting the boxes are labeled with data type names
  - Arrows indicate the flow of data between them

The Design of the Text-Analysis Program

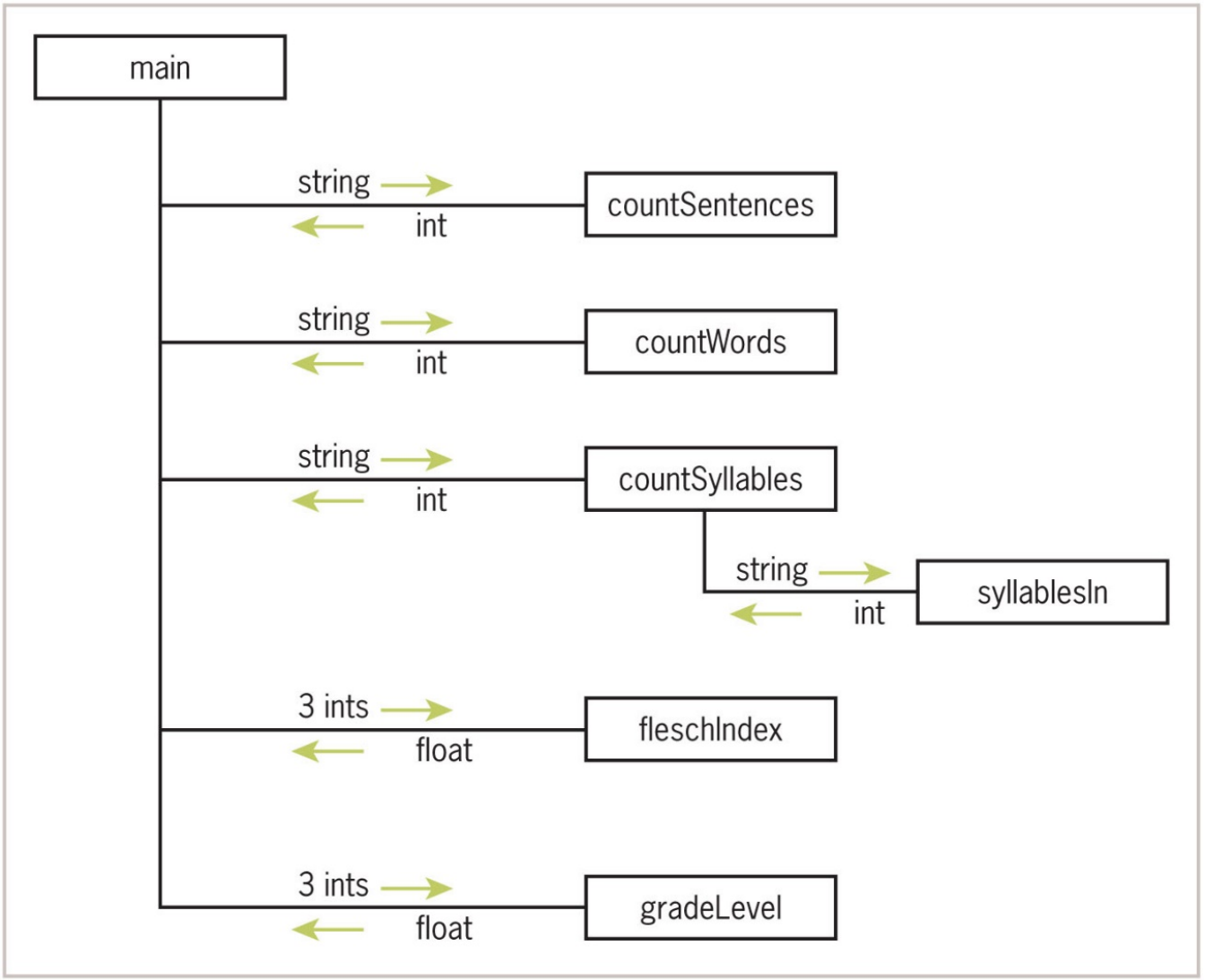


Figure 6-1 A structure chart for the text-analysis program

Defining a Main Function

- **main()** serves as the entry point for a script
  - Usually expects no arguments and returns no value
- Definition of main and other functions can appear in no particular order in the script
  - As long as main is called at the end of the script

```
def main():
    number = float(input("Enter a number: "))
    print("The square of", number, "is", square(number))
def square(x):
    return x * x

# The entry point for program execution
if __name__ == "__main__":
    main()
```

Recursive Functions

- In top-down design, decompose a complex problem into a set of simpler problems and solve with different functions
- In some cases, you can decompose a complex problem into smaller problems of the **same** form
  - Subproblems can be solved using the same function
  - Resulting functions are called **recursive functions**
- A **recursive** function is a function that calls itself
  - To prevent function from repeating itself indefinitely, it must contain at least one **selection** statement
  - Statement examines **base case** to determine whether to stop or to continue with another recursive step

UNIVERSITY OF MARYLAND

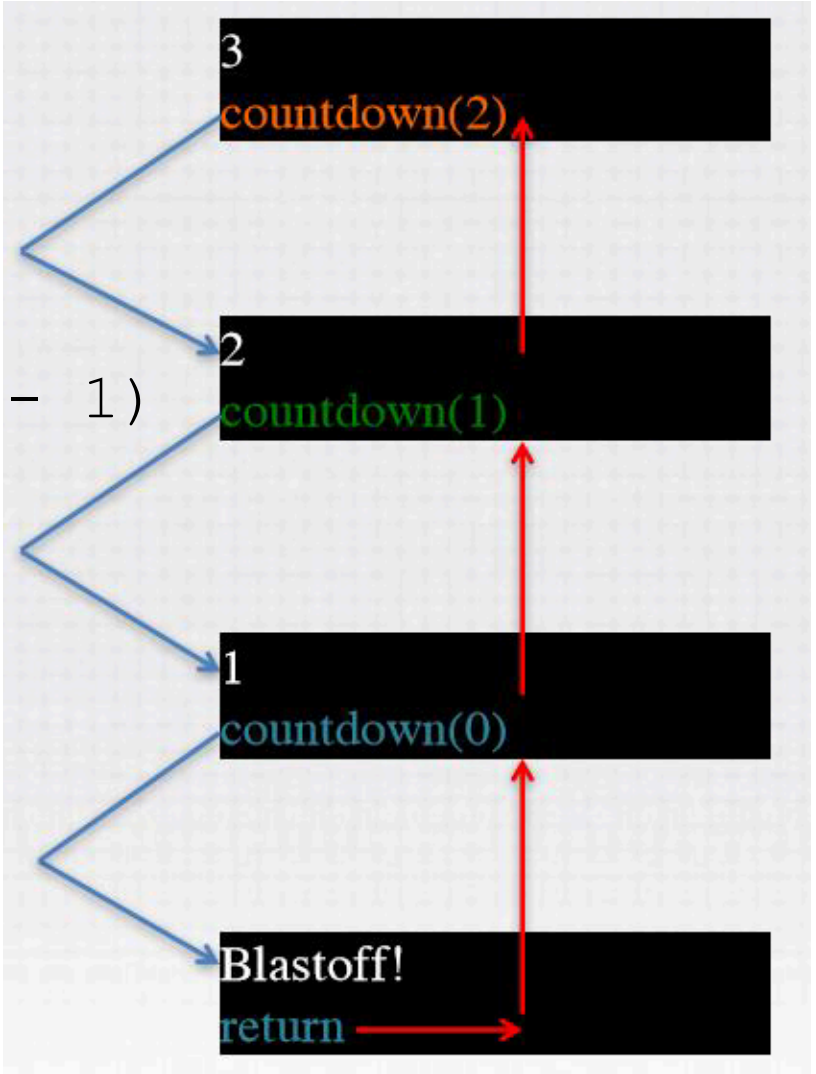
Recursion Example

How many rows in front of you?

- You: ask Amy sitting in front of you
- Amy: asks Bob sitting in front of her
- Bob: asks Cindy sitting in front of him
- …
- XXX: only one row in front of me
- (ZZZ: no rows in front of me)

Recursion Example: Countdown

```
def countdown(counter):
    if counter == 0:
        print("Blastoff!")
    else:
        print(counter)
        countdown(counter - 1)
>>> countdown(3)
3
2
1
Blastoff!
```



Convert Loop to Recursive Function

```
def displayRange(lower, upper):
    """Outputs the numbers from lower through
upper."""
    while lower <= upper:
        print(lower)
        lower = lower + 1
```

- You can replace loop with a **selection** statement and **assignment** statement with a **recursive call**

```
def displayRange(lower, upper):
    """Outputs the numbers from lower through
upper."""
    if lower <= upper:
        print(lower)
        displayRange(lower + 1, upper)
```

UNIVERSITY OF MARYLAND

Recursive Definitions to Recursive Functions

- A recursive definition consists of equations that state
  - what a value is for one or more base cases, and
  - one or more recursive cases
- Example: Factorials
  - n! = n * (n-1) * … * 1

```
def factorial(n):
    """Returns the product of an integer and all
integers below it."""
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
>>> factorial(10)
3628800
```

Recursive Definitions to Recursive Functions

- Example: Fibonacci sequence 1 1 2 3 5 8 13 ...
  - fib(n) = 1, when n = 1 or 2
  - fib(n) = fib(n−1) + fib(n−2), for all n > 2

```
def fib(n):
    """Returns the n-th Fibonacci number."""
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
>>> fib(1)
1
>>> fib(10)
55
```

Namespace

- A program's **namespace** is the set of its variables and their values
  - You can control it with good design principles
- **Module variables** and **temporary variables** receive their values as soon as they are introduced
- **Parameters** behave like a variable and are introduced in a function or method header
  - Do not receive a value until the function is called
- A **method** reference always uses an object followed by a **dot** and the method name

UNIVERSITY OF MARYLAND

Scope of Variables

- **Scope**: Area in which a name refers to a given value
- **Temporary/local variables** are restricted to the body of the functions in which they are introduced
- **Parameters** are invisible outside function definition
- The scope of **module/global variables** includes entire module below point where they are introduced
  - Although function can reference a module variable for its value, it cannot under normal circumstances assign a new value to a module variable
  - When such attempt is made, the PVM creates a new, temporary variable of the same name within the function

UNIVERSITY OF MARYLAND

Scope of Variables

```
global_variable = "Global"

def my_function(parameter="Parameter"):
    local_variable = "Local"
    print(global_variable)
    print(parameter)
    print(local_variable)

>>> my_function("Argument")
>>> my_function()

>>> print(global_variable)
>>> print(parameter)
>>> print(local_variable)
```

Lifetime

- Variable's **lifetime**: Period of time when variable has memory storage associated with it
  - When a variable comes into existence, storage is allocated for it; when it goes out of existence, storage is reclaimed by the PVM
- Module variables come into existence when introduced and generally exist for lifetime of program that introduces or imports them
- Parameters and temporary variables come into existence when bound to values during call, but go out of existence when call terminates

Example of Default Arguments

```
def repToInt(repString, base = 2):
    """Converts the repString to an int in the
base ..."""
    decimal = 0
    exponent = len(repString) - 1
    for digit in repString:
        decimal = decimal + int(digit) * base **
exponent
        exponent -= 1
    return decimal
>>> repToInt("10", 8) # Override the default to here
8
>>> repToInt("10", 2) # Same as the default, not
necessary
2
>>> repToInt("10") # Base 2 by default
2
```

UNIVERSITY OF MARYLAND

Example of Default Arguments Can Be Supplied in Two Ways: By position, or By keyword

```
def example(required, option1 = 2, option2 = 3):
    print(required, option1, option2)

>>> example(1) # Use all the defaults
1 2 3
>>> example(1, 10) # Override the first default
1 10 3
>>> example(1, 10, 20) # Override all the defaults
1 10 20
>>> example(1, option2 = 20) # Override the second
default
1 2 20
>>> example(1, option2 = 20, option1 = 10) # In
any order
1 10 20
```

UNIVERSITY OF MARYLAND

Example of Variable-Length Arguments

```
def printargs(arg, *vartuple):
    print("Arguments are:")
    print(arg)
    for var in vartuple:
        print(var)
    return None

>>> printargs(10)
Arguments are:
10
>>> printargs(10, 20, 30)
Arguments are:
10
20
30
```

Functions can be assigned to variables, passed as arguments, returned as values, and stored in data:

```
>>> abs # See what abs looks like
<built-in function abs>
>>> f = abs # f is an alias for abs
>>> f # Evaluate f
<built-in function abs>
>>> f(-4) # Apply f to an argument
4

>>> import math
>>> funcs = [abs, math.sqrt] # Put the functions
in a list
>>> funcs
[<built-in function abs>, <built-in function
sqrt>]
>>> funcs[1](2) # Apply math.sqrt to 2
1.4142135623730951
```

Mapping

- Mapping applies a function to each value in a sequence and returns a new sequence of the results

```
>>> words = ["231", "20", "-45", "99"]
>>> map(int, words) # Convert all strings to
ints
<map object at 0x14cbd90>
>>> words # Original list is not changed
['231', '20', '-45', '99']
>>> words = list(map(int, words)) # Reset
variable to change it
>>> words
[231, 20, -45, 99]
```

UNIVERSITY OF MARYLAND

Filtering

- When filtering, a function called a **predicate** is applied to each value in a list
  - If predicate returns True, value is added to a new list
  - otherwise, value is dropped from consideration

```
def odd(n):
    return n % 2 == 1

>>> list(filter(odd, range(10)))
[1, 3, 5, 7, 9]
```

Reducing

- When reducing, we take a list of values and repeatedly apply a function to accumulate a single data value

```
from functools import reduce
def add(x, y):
    return x + y
def multiply(x, y):
    return x * y

>>> data = [1, 2, 3, 4]
>>> reduce(add, data)
10
>>> reduce(multiply, data)
24
```

Lambda Functions

- A **lambda** is an **anonymous function**
  - When the lambda is applied to its arguments, its expression is evaluated and its value is returned
- The syntax of lambda is very tight and restrictive: lambda <argument−1, ..., argument−*n*>: <expression>
- All of the code must appear **on one line** and, although it is sad, a lambda cannot include a selection statement, because selection statements are not expressions

UNIVERSITY OF MARYLAND

Lambda Functions

- We can now specify addition or multiplication on the fly:

```
>>> data = [1, 2, 3, 4]
>>> reduce(lambda x, y: x + y, data) # Produce the sum
10
>>> reduce(lambda x, y: x * y, data) # Produce the
product
24
```

- Example shows the use of range, reduce, and lambda to simplify the definition of the summation function:

```
def summation(lower, upper):
    if lower > upper:
        return 0
    else:
        return reduce(lambda x, y: x + y, range(lower,
upper + 1))
```