Functions

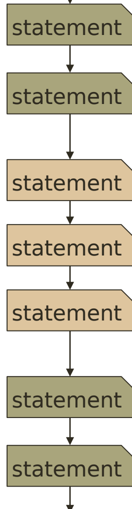
zyBooks
"Programming in Python 3"
chapter 6

Why a Function?

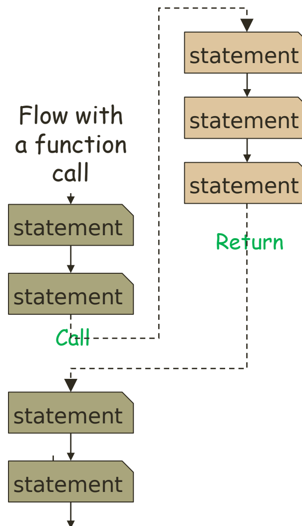
- Some groups of statements work together to achieve a specific goal
- Some groups of statements occur more than once

- A *function* is a group of statements that cooperate to produce a defined result
  - Often they calculate a value or values and return them
    - predefined examples: the **`cos()`** function, the **`sum()`** function, the **`range()`** function, etc.

Function Calls



- Instead of doing each statement in order…
- Related statements are grouped into a *function*…
- That is *called* from the main flow…
- And *returns* to wherever it was called from when done.



Sources of Functions

- Standard functions
  - len()
  - sum()
  - range()
- Imported from a module
  - os.listdir()
  - random.randrange()
  - time.ctime()
  - shutil.get_terminal_size()
- User-defined
  - *Whatever you need to do…*

User-Defined Functions

- *Define* a function with a *parameter* list, and statements that do the work
- Two ways to make/use a function:
  - A function that does some calculation, and returns a result
    - Like a mathematical function
    - Trivial example:
      - ```
        def average(valueA, valueB , valueC):
            return (valueA + valueB + valueC) / 3
        ```
  - A "standalone" function that does some input and/or output, does something, and is done
    - "main()" is used like this

Defining a Function

- *Define* a function with a *parameter* list, and statements that do the work
- An almost-as-trivial example:
  - ```
    def geometric_mean(valueList):
        import math
        return (math.prod(valueList) ** (1/len(valueList))
    ```
- Place the definition before the first use of the function within the program
  - Outside of other functions, or inside another one
- *Call* the function by specifying its name along with a list of values (or *arguments*) that serve for its parameters
  - `gm = geometric_mean( [17, -29.3, 100, 45.101010] )`

Parameters Can Be Any Type

- Easy calculation that depends on two other, standard functions

```
def mean( datalist ):
    m = sum(datalist) / len(datalist)
    return m
#----
```

Example: Hailstone-Sequence Function

- x_n is generated by:
  - $x_n = \begin{cases} \frac{x_{n-1}}{2}, \text{ if } x_{n-1} \text{ even} \\ (3 \times x_{n-1}) + 1, \text{if } x_{n-1} \text{ odd} \end{cases}$

  with arbitrary positive starting value x_0
- Function that generates x_n :

```
def hailstone( x ):
    if  x % 2 == 0:     # is x even?
        next_x = x/2
    else:
        next_x = (3*x) + 1
    return next_x
```

Using hailstone()

- Program that calls `hailstone()` function:
  -

```
hs = input('Starting value? ')
length = 1
while hs != 1:
    hs = hailstone(hs)
    length += 1
    print(hs)
print("Sequence is",length, "items long")
```

hailstone(), with formatting

- Make the output look a bit nicer:
  -

```
hs = input('Starting value? ')
length = 1
while hs != 1:
    hs = hailstone(hs)
    length += 1
    print( "{:6d}".format(hs) )
print("Sequence is {:d} items long" \
    .format(length) )
```

Review

- What is the *parameter* of the hailstone() function?
- What is the *return value* of the hailstone() function?
- Rewrite hailstone():
  - ```
    def hailstone( x ):
        if  x % 2 == 0:     # is x even?
            return x/2
        else:
            return (3*x) + 1
    ```
- Is this version better?

Review 2

- Rewrite hailstone():
  - ```
    def hailstone( x ):
        # is x even?
        return x/2 if x % 2 == 0 else (3*x) + 1
    ```
- Is this version better?
- Which version is the easiest for *you* to understand?

Dynamic Parameter Types *(advanced issue)*

- Functions can accept parameters of any type…
  - It depends on what the function does
- An *individual* function can have a parameter whose type changes from call to call…
  - What the function does should depend on the type of the parameter
- This is called *polymorphism*
  - "Multiple shapes", *i.e.* one function name does different things with different types of parameter
- The function will need to determine the arguments' types each time it is called

A Polymorphic Function

- This function joins two strings, or joins a number and a string, or adds two numbers:
  -

```
def polysum(A, B):
    if type(A) == type('') and \
                     type(B) == type(''):
        return A + B
    elif type(A) == type(''):
        return A + str(B)
    elif type(B) == type(''):
        return str(A) + B
    else:
        return A + B
```

Stubs

- Sometimes you know how to do most of a program, but one or two steps are tricky
- Temporarily replace tricky parts with a *stub* that allows the program to keep going
- Simple stubs:
  - Replace an if- or else-clause with a single **pass** statement
  - Replace s function body with a statement that returns a plausible value for testing purposes

Example: Descriptive-Statistics Functions

  - Single-value summaries of potentially-large datasets
  - Work on a list of numeric values
- sum(), len(), min(), max()
  - standard (built-in) functions
- span()
  - equal to (max() – min())
- mean()
  - also known as "average"
- standard deviation, variance
  - measure of "spread" of the data
- median()
  - "middle" value of the dataset
    - half of values below, half above

Mean, again

- Easy calculation depends on two other, standard functions

```
def mean( datalist ):
    m = sum(datalist) / len(datalist)
    return m
#----
```

Median

- Find the middle value of a *sorted* list of values
- If odd number of elements in list:
  - integer division of list's length by two gives index of middle element
- If even number of elements:
  - no middle element, return average of two central elements instead
  - integer division of list's length by two gives "upper" of two central indexes
- Different ways to decide if "odd" or "even"
  a) x // 2 == x / 2
  b) x % 2 == 0   # more-common way to do this

Median function

```
def median(datalist):
    dl = sorted(datalist)
    l = len(dl)
    iUpper = l // 2
    if l % 2 == 0:
        iLower = iUpper - 1
        m = (dl[iLower] + dl[iUpper]) / 2
    else:
        m = dl[iUpper]
    return m
#----
```

Example: average, median of a datafile

- Trivial program to read numbers from a disk file, then do statistics on them:

```
#!/usr/bin/env python3

def read_file(filename):
    pass      # Do something here...
    return [2,3,5,7]      # fake data for testing

datalist = read_file( 'mydata.txt' )
if len(datalist) == 0:
    pass      # what to do about missing data?
else:
    print(mean(datalist), median(datalist))
```

Variance and Standard Deviation

- Measures of "dispersion" of a collection of values
- Standard deviation is the square root of variance
  - s.d. is more useful than variance
  - variance is easier to calculate

Calculating the Variance - Formula 1

- Compute squares of the data
- Calculate the sum of the data, the sum of the squares of the data, and the count *n*
- Calculate the *Variance* as:

$$\text{variance} = \frac{\left(\sum x^2 - \frac{(\sum x)^2}{n}\right)}{(n-1)}$$

Observation: this calculation only depends on the sums and count, not on the individual data points. It can be updated as more data points are added.

Calculating the Standard Deviation

- The variance is a good measure of the scattering of the data, but...
  - Consider the units of the original data?
  - Consider the units of the mean?
  - Consider the units of the differences?
  - Consider the units of the variance???
- The units of the variance aren't directly comparable to the original data units.
- The *Standard Deviation* is the square root of the variance.
  - It has the same units as the original data and the mean.

Variance and Standard Deviation

```
def variance( datalist ):
    n = len(datalist)
    squares = [ x**2 for x in datalist ]
    ssq = sum(squares)
    sqs = (sum(datalist))**2
    v = (ssq - sqs/n) / (n-1)
    return v
#----
```

```
def stdev( datalist ):
    return variance(datalist)**0.5
#----
```

Calculating the Variance – Another Way

- Compute the mean (the average) of the data
- Create a column of *differences* between each data point and the mean
  - These are the *errors*
- Calculate the squares of the differences
  - These are the Squared Errors
- Find the sum of the Squared Errors
  - abbreviated SSE
- Divide the SSE by (n-1) to get the *Variance*
  - where n is the number of data points
  - (n-1) is called the *degrees of freedom*

Formula 2 for Variance – the Sum of Square Errors, SSE

$$\text{SSE} = \sum (x_i - \mu)^2$$

$$\text{variance} = \frac{\text{SSE}}{(n-1)}$$

- where
  - » x_i are the individual data points
  - » μ is the mean (the average) of all the data points
  - » n is the number of data points

Variance 2

```
def variance( datalist ):
    n = len(datalist)
    m = mean(datalist)
    se = [ (x - m)**2 for x in datalist ]
    sse = sum(se)
    v = sse / (n - 1)
    return v
#----
```

```
def stdev( datalist ):
    return variance(datalist)**0.5
#----
```

try

- Generate or read in a list of numbers
- Write and apply stats functions to the list

Statistics Functions and Modules

- Statistics functions can be used many times
- Collect functions together for reuse
- *Module* contains related functions and variables
- Comments in the form of triple-quoted strings serve as *documentation* for items in the module (and for the module itself)

Module and Function Documentation

```
#!/usr/bin/env python3
''' Some basic descriptive-statistics functions.
Variance can be calculated in two ways, named as
varianceA and varianceB.  Standard deviation is
based on varianceB.
2016-11-04
'''

def variance( datalist ):
    '''Variance of a list of numbers, calculated according
    to the definition of variance.'''
    n = len(datalist)
    m = mean(datalist)
    se = [ (x - m)**2 for x in datalist ]
    sse = sum(se)
    v = sse / (n - 1)
```