BIOS6642

Functions

Fuyong Xing

Department of Biostatistics and Informatics

Colorado School of Public Health

University of Colorado Anschutz Medical Campus

Outline

- **Function basics**
- Parameters and others
- Documenting functions
- Incremental development
- Functions examples

Function basics

- The codes on the right compute the greatest common divisor (GCD) of two integers.
- If we want to compute the GCD from several different places within our program, do we need to copy the codes to multiple places?

```
# Compute the greastest common factor of two integers
# provided by the user

# Prompt user for input
num1 = int(input('Please enter an integer: '))
num2 = int(input('Please enter another integer: '))

# Determine the smaller of num1 and num2
min = num1 if num1 < num2 else num2

# 1 definitely is a common factor to all ints
largest_factor = 1
for i in range(1, min + 1):
    if num1 % i == 0 and num2 % i == 0:
        largest_factor = i   # Found larger factor
# Print the GCD
print(largest_factor)
```

Function basics

- We can define a **function** to package the codes and make them more reusable, instead of copying and pasting.

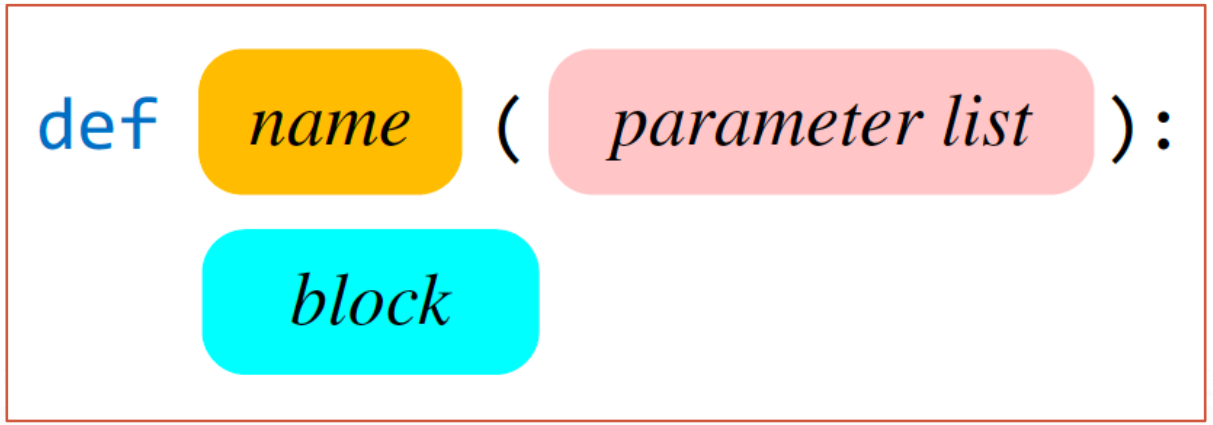
```
def gcd(num1, num2):
    # Determine the smaller of num1 and num2
    min = num1 if num1 < num2 else num2
    # 1 definitely is a common factor to all ints
    largest_factor = 1
    for i in range(1, min + 1):
        if num1 % i == 0 and num2 % i == 0:
            largest_factor = i   # Found larger factor
    return largest_factor
```

Function basics

- A **function** is a named sequence of statements that performs some useful operation.
- Why functions?
  - Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
  - Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
  - Dividing a long program into functions allows you to debug the parts one at a time and then assemble them.
  - Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Function basics

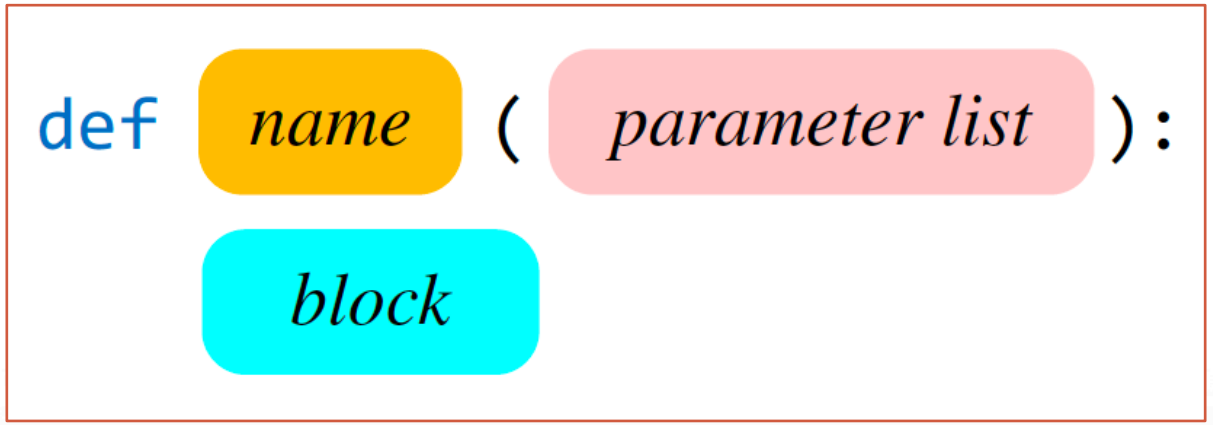
- **Function definition**: contains the code that determines the function's behavior.



- The *def* keyword introduces a function definition.
- The function *name* has the same rule as variable names.
- The *parameters* appear in a parenthesized comma-separated list.
- The *body* is a block of indented statements to execute when callers invoke the function.

Function basics

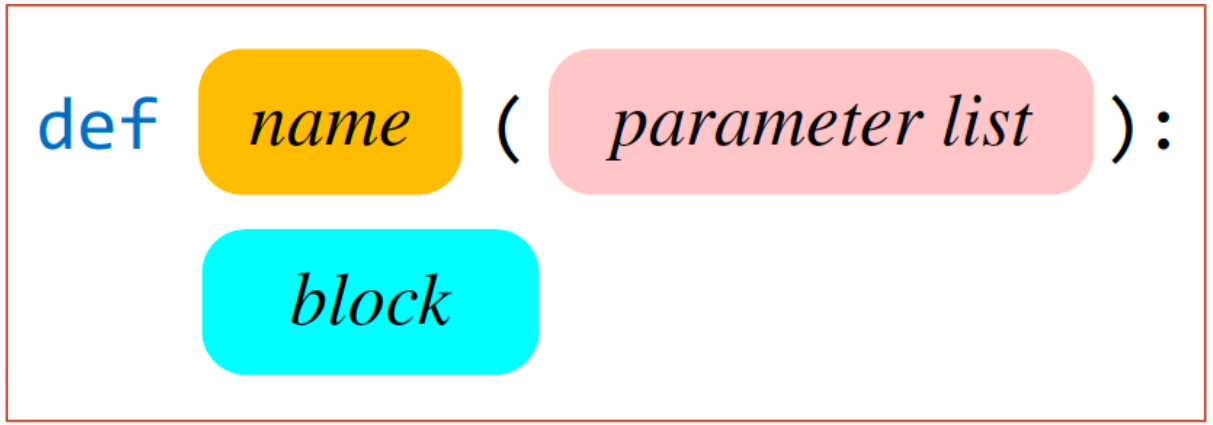
- **Function definition**: contains the code that determines the function's behavior.



- **Function invocation**: A function is used within a program via a function invocation or call.

Function basics

- **Function definition**: contains the code that determines the function's behavior.



- **Function invocation**: A function is used within a program via a function invocation or call.
- Every function has exactly one definition but may have multiple invocations.
- The code within a function definition executes only when invoked by a caller.

Function basics

- A simple example

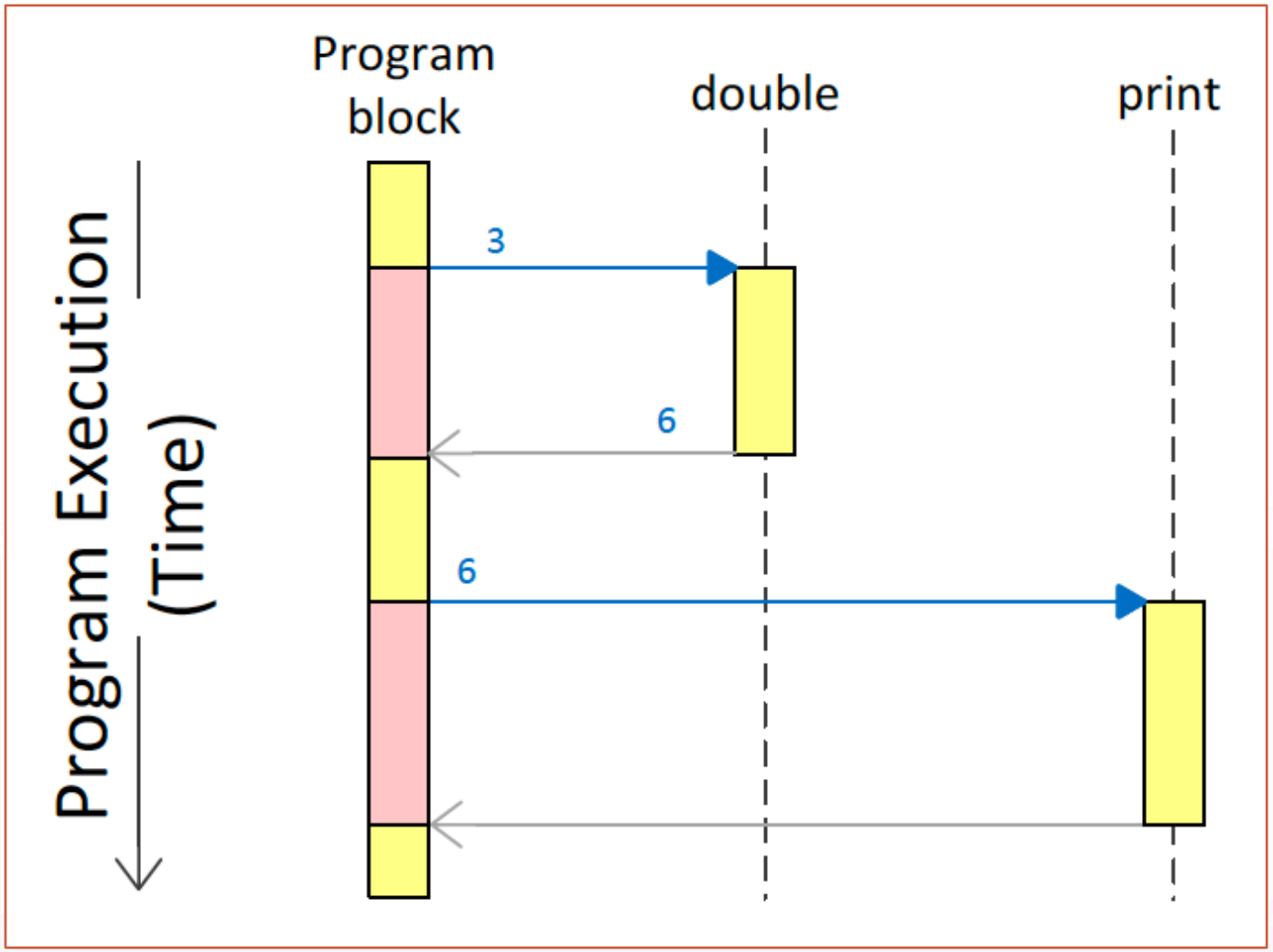
n is a formal parameter

```
def double(n):
    return 2 * n  # Return twice the given number

# Call the function with the value 3 and print its result
x = double(3)
print(x)
```

3 is an argument or actual parameter

Function basics

- The diagram on the right illustrates the execution of the code.
- A function invocation binds the actual parameters sent by the caller to their corresponding formal parameters.



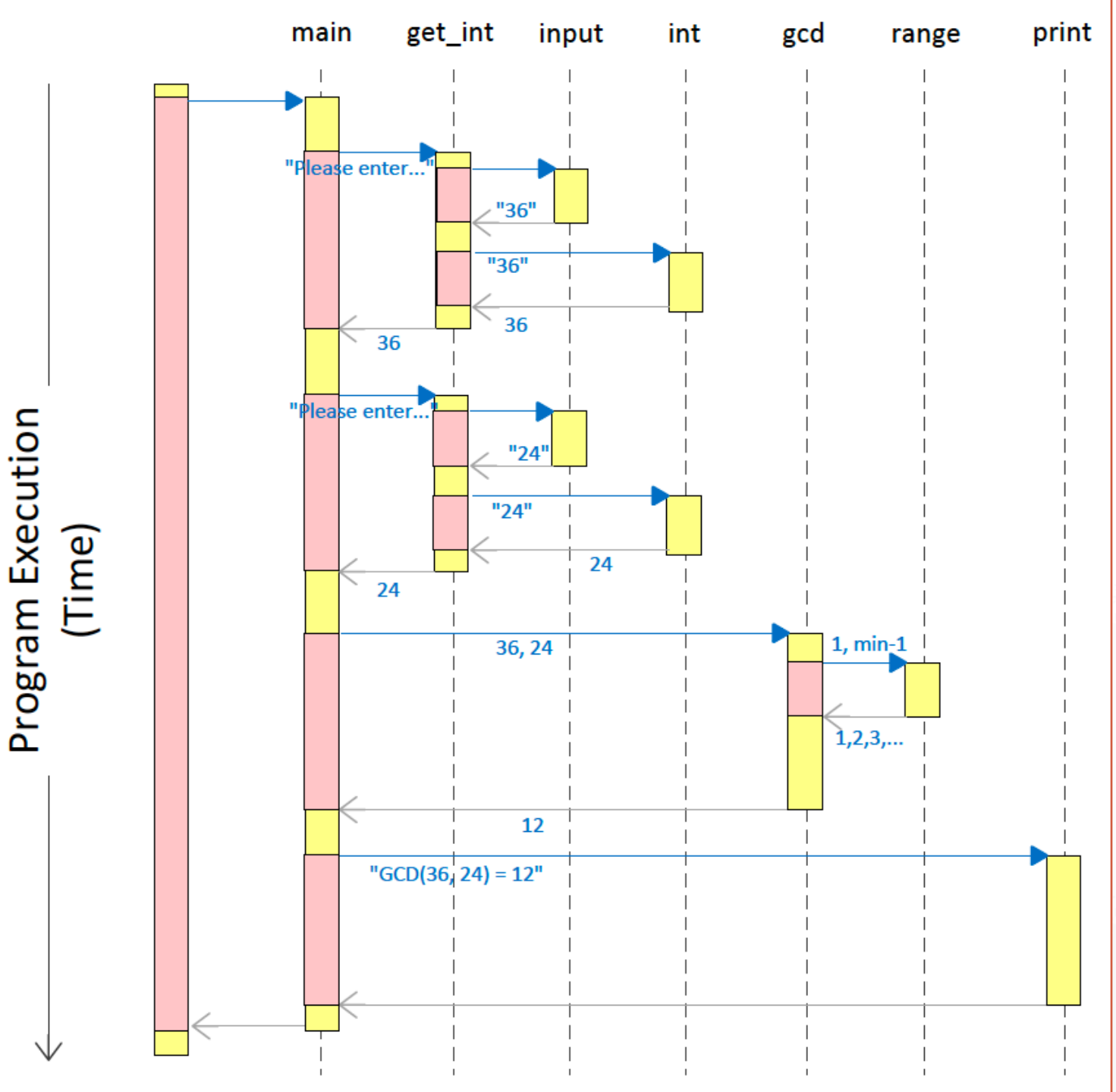
It has 2 parameters

It has no parameters

```
def gcd(n1, n2):
    # Determine the smaller of n1 and n2
    min = n1 if n1 < n2 else n2
    # 1 definitely is a common factor to all ints
    largest_factor = 1
    for i in range(1, min + 1):
        if n1 % i == 0 and n2 % i == 0:
            largest_factor = i   # Found larger factor
    return largest_factor

#  Get an integer from the user
def get_int():
    return int(input("Please enter an integer: "))


#  Main code to execute
def main():
    n1 = get_int()
    n2 = get_int()
    print("gcd(", n1, ",",  n2, ") = ", gcd(n1, n2), sep="")


# Run the program
main()
```

Another example

The diagram for code execution



Outline

- Function basics
- **Parameters and others**
- Documenting functions
- Incremental development
- Functions examples

Parameters and others

- Parameter passing: The function call binds to the formal parameter the object referenced by the actual parameter.

Codes

```
def increment(x):
    print("Beginning execution of increment, x =", x)
    x += 1    # Increment x
    print("Ending execution of increment, x =", x)


def main():
    x = 5
    print("Before increment, x =", x)
    increment(x)
    print("After increment, x =", x)


main()
```

Parameters and others

- Parameter passing: The function call binds to the formal parameter the object referenced by the actual parameter.
- Objects with type of integer, floating-point number or string are immutable.

Output

```
Before increment, x = 5
Beginning execution of increment, x = 5
Ending execution of increment, x = 6
After increment, x = 5
```

Parameters and others

- We can pass literals, variables, or expressions as actual parameters.
  - *gcd(24,36)*
  - *gcd(x,36)* # x is a variable
  - *gcd(x-2, gcd(10,8))* # pass the results of gcd(10,8) as the second actual parameter

```
def gcd(n1, n2):
    # Determine the smaller of n1 and n2
    min = n1 if n1 < n2 else n2
    # 1 definitely is a common factor to all ints
    largest_factor = 1
    for i in range(1, min + 1):
        if n1 % i == 0 and n2 % i == 0:
            largest_factor = i   # Found larger factor
    return largest_factor
```

Parameters and others

- During a function call, the numbers of formal and actual parameters are equal.

Parameters and others

- During a function call, the numbers of formal and actual parameters are equal.
- We can allow a function to accept a varying number of parameters by using **default parameters**. The default values will be used if no corresponding actual parameters are provided during a function call.

n has a default value as 10.

```
def countdown(n=10):
    for count in range(n, -1, -1):  # Count down from n to zero
        print(count)
```

- *countdown()* prints 10,9,8,7,6,5,4,3,2,1,0.
- *countdown(5)* prints 5,4,3,2,1,0.

Parameters and others

- If a function has both non-default and default parameters, all default parameters within the parameter list must appear after all the non-default parameters.

```
def sum_range(n, m=100):      # OK, default follows non-default
    sum = 0
    for val in range(n, m + 1):
        sum += val


def sum_range(n=0, m=100):      # OK, both default
    sum = 0
    for val in range(n, m + 1):
        sum += val


def sum_range(n=0, m):    # Illegal, non-default follows default
    sum = 0
    for val in range(n, m + 1):
        sum += val
```

Parameters and others

- Variables defined in a function are local variables.
- The scope of a local variable is the point within the function's block after its first assignment until the end of that block.

```
x = 2
print("1. x =", x)  # Print variable x's current value

def fun1():
    x = 10
    print("2. x =", x)  # Print this variable x's current value

print("3. x =", x)  # Print variable x's current value

def fun2():
    x = 20
    print("4. x =", x)  # Print this variable x's current value

print("5. x =", x)  # Print variable x's current value
fun1()  # Invoke function fun1
fun2()  # Invoke function fun2

print("6. x =", x)  # Print variable x's current value
```

Codes

Parameters and others

- Variables defined in a function are local variables.
- The scope of a local variable is the point within the function's block after its first assignment until the end of that block.

```
1. x = 2
3. x = 2
5. x = 2
2. x = 10
4. x = 20
6. x = 2
```

Output

Parameters and others

- A function may not have a *return* statement, and this function will return the special object *None*.

```
#  Count to ten and print each number on its own line
def count_to_10():
    for i in range(1, 11):
        print(i, end=' ')
    print()


print("Going to count to ten . . .")
count_to_10()
print("Going to count to ten again. . .")
count_to_10()
```

Parameters and others

- A function can return multiple values.
  - Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values.

```
def midpoint(pt1, pt2):
    x1, y1 = pt1    # Extract x and y components from the first point
    x2, y2 = pt2    # Extract x and y components from the second point
    return (x1 + x2)/2, (y1 + y2)/2
```

Outline

- Function basics
- Parameters and others
- **Documenting functions**
- Incremental development
- Functions examples

Documenting functions

- Documenting a function's definition is helpful for programmers to use or extend the function.
- Important information to be documented:
  - The purpose of the function
  - The role of each parameter
  - The nature of the return value

Documenting functions

- **Docstring**: a string literal appears as the first statement in the block of a function definition.

```
def gcd(n1, n2):
    """ Computes the greatest common divisor of integers n1 and n2.  """
    # Determine the smaller of n1 and n2
    min = n1 if n1 < n2 else n2
    # 1 definitely is a common factor to all ints
    largest_factor = 1
    for i in range(1, min + 1):
        if n1 % i == 0 and n2 % i == 0:
            largest_factor = i   # Found larger factor
    return largest_factor
```

Documenting functions

- Can access the docstring by using
  - Doc attribute: *print(gcd.__doc__)*
  - The *help* function: *help(gcd)*

```
>>> help(gcd)
Help on function gcd in module docgcd:

gcd(n1, n2)
    Computes the greatest common divisor of integers n1 and n2.
```

Documenting functions

- The following information can also be included in the documentation (but not in the docstring).
  - Author of the function
  - Date that the function's implementation as last modified
  - Reference: If the code is adapted from another source, please list the source. The reference may consist of a Web URL.

Documenting functions

- Example

```
#       Author: Joe Algori (joe@eng-sys.net)
#       Last modified:  2010-01-06
#       Adapted from a formula published at
#       http://en.wikipedia.org/wiki/Distance
def distance(x1, y1, x2, y2):
    """
       Computes the distance between two geometric points
         x1 is the x coordinate of the first point
         y1 is the y coordinate of the first point
         x2 is the x coordinate of the second point
         y2 is the y coordinate of the second point
       Returns the distance between (x1,y1) and (x2,y2)
    """
    ...
```

Outline

- Function basics
- Parameters and others
- Documenting functions
- **Incremental development**
- Functions examples

Incremental development

- Incremental development: avoid long debugging sessions by adding and testing only a small amount of code at a time.
- Suppose we want to write a function to calculate the distance between two points, (x_1, y_1) and (x_2, y_2).

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Incremental development

- Step 1: write an outline of the function and test it to ensure it is syntactically correct

Definition

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Test

```
>>> distance(1, 2, 4, 6)
0.0
```

Incremental development

- Step 2: find difference between two points and test

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

Incremental development

- Step 3: compute the sum of squares and test

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

Incremental development

- Step 4: use a Python function *math.sqrt* in the standard library to compute and the return the result, and test.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

Incremental development

- Summary of incremental development:
  1. Start with a working program and make small incremental changes. At any point, if there is an error, you should have a good idea where it is.
  2. Use variables to hold intermediate values so you can display and check them.
  3. Once the program is working, you might want to remove some of the scaffolding codes or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

Outline

- Function basics
- Parameters and others
- Documenting functions
- Incremental development
- **Functions examples**

This program prints a sequence of prime numbers.

```
from math import sqrt

def is_prime(n):
    """
    Determines the primality of a given value.
    n an integer to test for primality.
    Returns true if n is prime; otherwise, returns false.
    """
    root = round(sqrt(n)) + 1
    # Try all potential factors from 2 to the square root of n
    for trial_factor in range(2, root):
        if n % trial_factor == 0:  # Is it a factor?
            return False           # Found a factor
    return True                    # No factors found


def main():
    """
    Tests for primality each integer from 2 up to a value provided by the user.
    If an integer is prime, it prints it; otherwise, the number is not printed.
    """
    max_value = int(input("Display primes up to what value? "))
    for value in range(2, max_value + 1):
        if is_prime(value):          # See if value is prime
            print(value, end=" ")    # Display the prime number
    print()  # Move cursor down to next line


main()   # Run the program
```

Floating-point equality testing

```
from math import fabs

def equals(a, b, tolerance):
    """
    Returns true if a = b or |a - b| < tolerance.
    If a and b differ by only a small amount (specified by tolerance), a and b are considered
    "equal."  Useful to account for floating-point round-off error.
    The == operator is checked first since some special floating-point values such as
    floating-point infinity require an exact equality check.
    """
    return a == b or fabs(a - b) < tolerance


def main():
    """ Try out the equals function """
    i = 0.0
    while not equals(i, 1.0, 0.0001):
        print("i =", i)
        i += 0.1


main()
```

Readings

- Think Python, chapters 3 and 6.1 ~ 6.4

References

- *Think Python: How to Think Like a Computer Scientist*, 2nd edition, 2015, by Allen Downey
- *Python Cookbook*, 3rd edition, by David Beazley and Brian K. Jones, 2013
- *Python for Data Analysis*, 2nd edition, by Wes McKinney, 2018
- *Fundamentals of Python Programming,* by Richard L. Halterman