

6.1 User-defined function basics

Function basics

A **function** is a named series of statements.

- A **function definition** consists of the new function's name and a block of statements. Ex: `def print_pizza_area():`. An indented block of statements follows the definition.
- A **function call** is an invocation of the function's name, causing the function's statements to execute.

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Python comes with a number of built-in functions, such as `input()`, `int()`, `len()`, etc. The **def** keyword is used to create new functions.

The function call `print_pizza_area()` in the animation below causes execution to jump to the function's statements. Execution returns to the original location after executing the function's last statement.

Good practice is to follow the convention of naming functions with lowercase letters and underscores, such as `print_names` or `print_face`.

Other aspects of the function definition, like the `()`, are discussed later.

PARTICIPATION ACTIVITY

6.1.1: Function example: Printing a pizza area.



Animation content:

undefined

Animation captions:

1. The function call jumps execution to the function's statements.
2. After the last statement, execution returns to the original location.

PARTICIPATION ACTIVITY

6.1.2: Function basics.



Given the following program and the `print_pizza_area()` function defined above:

```
print_pizza_area()
print_pizza_area()
```

- 1) How many function calls to `print_pizza_area()` exist?

Check Show answer

- 2) How many function definitions of `print_pizza_area()` exist?

Check Show answer

- 3) How many output statements would execute in total?

Check Show answer

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

4) How many print statements exist in the program code?

Check

Show answer

**PARTICIPATION
ACTIVITY**

6.1.3: Calling a function.

```
1 def print_face():
2     face_char = 'o'
3     print(' ', face_char, ' ', face_char) # Print eyes
4     print('   ', face_char)              # Print nose
5     print(' ', face_char*5)              # Print mouth
6
7 print('Say cheese!')
8
9 print_face()
10
11 print('Did it turn out ok?')
```

<< First < Back Step 1 of 10 Forward > Last >>

→ line that has just executed

→ next line to execute

Frames

Objects

**CHALLENGE
ACTIVITY**

6.1.1: Basic function call output.

print_pattern() prints 5 characters. Call print_pattern() twice to print 10 characters. Example output:

```
*****
*****
```

```
1 def print_pattern():
2     print('*****')
3
4     ''' Your solution goes here '''
5     |
```

Run

View solution ▼ (Instructors only)

Download student submissions

**CHALLENGE
ACTIVITY**

6.1.2: Basic function call.

Define `print_shape()` to print the below shape. Example output:

```
***
***
***
```

```
1
2 ''' Your solution goes here '''
3
4 print_shape()
```

Run

[View solution](#) ▼ *(Instructors only)*

[Download student submissions](#)

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

6.2 Function parameters

Parameters

A programmer can influence a function's behavior via an input.

- A **parameter** is a function input specified in a function definition. Ex: A pizza area function might have `diameter` as an input.
- An **argument** is a value provided to a function's parameter during a function call. Ex: A pizza area function might be called as `print_pizza_area(12.0)` or as `print_pizza_area(16.0)`.

A parameter is like a variable definition. Upon entering the function, the parameter is bound to the argument object provided by the call, creating a shared reference to the object. Upon return, the parameter can no longer be used.

An argument may be an expression, like `12.0`, `x`, or `x * 1.5`.

**PARTICIPATION
ACTIVITY**

6.2.1: Function parameters.

```
1 def print_pizza_area(pizza_diameter):
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

```

2 pi_val = 3.14159265
3 pizza_radius = pizza_diameter / 2.0
4 pizza_area = pi_val * pizza_radius * pizza_radius
5 print('{:.1f} inch pizza is {:.3f} inches squared'
6       .format(pizza_diameter, pizza_area))
7
8 print_pizza_area(12.0)
9 print_pizza_area(16.0)

```

<< First < Back Step 1 of 17 Forward > Last >>

→ line that has just executed
 → next line to execute

Frames Objects

zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020

PARTICIPATION ACTIVITY 6.2.2: Parameters.

- 1) Complete the function definition to have a parameter named `user_age`.

```
def print_age( ): :
```

Check Show answer

- 2) Call a function named `print_age`, passing the value 21 as an argument.

Check Show answer

- 3) Is the following a valid function definition beginning? Type yes or no.
`def my_fct(userNum + 5):`

Check Show answer

- 4) Assume a function `def print_num(user_num):` simply prints the value of `user_num` without any space or newline. What will the following code output?

```
print_num(43)
print_num(21)
```

Check Show answer

zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020

Multiple parameters

A function may have multiple parameters, which are separated by commas. Argument values are assigned to parameters by position: First argument to the first parameter, second to the second, etc.

A function definition with no parameters must still have the parentheses, as in: `def print_something():`. The call to such a function must include parentheses, and they must be empty, as in: `print_something()`.

Figure 6.2.1: Function with multiple parameters

```
def print_pizza_volume(pizza_diameter, pizza_height):
    pi_val = 3.14159265

    pizza_radius = pizza_diameter / 2.0
    pizza_area = pi_val * pizza_radius * pizza_radius
    pizza_volume = pizza_area * pizza_height
    print('{:.1f} x {:.1f} inch pizza is {:.3f} inches cubed.'
          .format(pizza_diameter, pizza_height, pizza_volume))
```

```
print_pizza_volume(12.0, 0.3)
print_pizza_volume(12.0, 0.8)
print_pizza_volume(16.0, 0.8)
```

```
12.0 x 0.3 inch pizza is 33.929 inches cubed.
12.0 x 0.8 inch pizza is 90.478 inches cubed.
16.0 x 0.8 inch pizza is 160.850 inches cubed.
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

PARTICIPATION ACTIVITY

6.2.3: Multiple parameters.

- 1) Which correctly defines two parameters
x and y for a function definition:

`def calc_val(...):?`

- ☐ (x; y)
- ☐ (x y)
- ☐ (x, y)

- 2) Which correctly passes two integer
arguments for the function call

`calc_val(...)?`

- ☐ (99, 44 + 5)
- ☐ (99 + 44)
- ☐ (99 44)

- 3) Given a function definition:

`def calc_val(a, b, c):`

what value is assigned to b during this
function call:

`calc_val(42, 55, 77)?`

- ☐ Unknown
- ☐ 42
- ☐ 55

- 4) Given a function definition:

`def calc_val(a, b, c):`

and given variables i, j, and k, which are
valid arguments in the call

`calc_val(...)?`

- ☐ (i, j)
- ☐ (k, i + j, 99)
- ☐ (i + j + k)

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

PARTICIPATION ACTIVITY

6.2.4: Calls with multiple parameters.

Given:

```
def print_sum(num1, num2):
    print(num1, '+', num2, 'is', (num1 + num2))
```

- 1)

What will be printed for the following function call?

```
print_sum(1, 2)
```

Check

Show answer

- 2) Write a function call using `print_sum()` to print the sum of `x` and 400 (providing the arguments in that order).

Check

Show answer



©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

CHALLENGE
ACTIVITY

6.2.1: Function parameters.



Start

Type the program's output.

```
def print_age(user_age):  
    print('He is {}'.format(user_age))  
  
age_to_print = 19  
print_age(age_to_print)
```

He is 19

1

2

3

4

Check

Next

View solution ▼ (Instructors only)



CHALLENGE
ACTIVITY

6.2.2: Basic function call.



Complete the function definition to output the hours given minutes.

Sample output with input: 210.0

3.5

```
1 def output_minutes_as_hours(orig_minutes):  
2  
3     ''' Your solution goes here '''  
4  
5 minutes = float(input())  
6 output_minutes_as_hours(minutes)
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Run

View solution ▼ *(Instructors only)*

[↓ Download student submissions](#)
ⓘ

CHALLENGE ACTIVITY

6.2.3: Function call with parameters: Converting measurements.

Define a function `print_total_inches`, with parameters `num_feet` and `num_inches`, that prints the total number of inches. Note: There are 12 inches in a foot.

Sample output with inputs: 5 8

Total inches: 68

```

1
2 ''' Your solution goes here '''
3
4 feet = int(input())
5 inches = int(input())
6 print_total_inches(feet, inches)

```

Run

View solution ▼ *(Instructors only)*

[↓ Download student submissions](#)
ⓘ

@zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020

6.3 Returning values from functions

Return statements

A function may return one value using a **return statement**. Below, the `compute_square()` function returns the square of its argument.

@zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020

PARTICIPATION ACTIVITY

6.3.1: Function example: Returning a value.

Animation content:

undefined

Animation captions:

1. Call `compute_square` and pass in the value 7.
2. Compute the square of `num_to_square` and return the result.
3. `num_squared` is assigned the return value of `compute_square(7)`.

A function can only return one item, not two or more (though a list with multiple elements could be returned). A function with no return statement, or a return statement with no following expression, returns the value **None**. **None** is a special keyword that indicates no value.

A return statement may appear at any point in a function, not just as the last statement. A function may also contain multiple return statements in different locations.

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSC CSE20NawabWinter2020

PARTICIPATION ACTIVITY

6.3.2: Return basics.

- 1) Add a return statement to the function that returns the result of adding `num1` and `num2`.

```
def sum(num1, num2):  
    return
```

Check Show answer

- 2) Add a return statement to the function that returns the cube of the argument, i.e. (`num*num*num`).

```
def cubed(num):  
    
```

Check Show answer

Calling functions in expressions

A function evaluates to its returned value. Thus, a function call often appears within an expression. Ex:

`5 + compute_square(4)` would become `5 + 16`, or `21`. A function that returns **None** cannot be used as such within an expression.

PARTICIPATION ACTIVITY

6.3.3: Function calls in an expression.

Given the following function, determine which statements are valid.

```
def square_root(x):  
    return math.sqrt(x)  
  
def print_val(x):  
    print(x)
```

- 1) `y = square_root(49.0)`
☐ True
☐ False
- 2) `square_root(49.0) = z`
☐ True
☐ False
- 3) `y = 1.0 + square_root(144.0)`

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSC CSE20NawabWinter2020

- True
☐ False
- 4) `y = square_root(square_root(16.0))`
☐ True
☐ False
- 5) `y = square_root()`
☐ True
☐ False
- 6) `square_root(9.0)`
☐ True
☐ False
- 7) `y = print_val(9.0)`
☐ True
☐ False
- 8) `y = 1 + print_val(9.0)`
☐ True
☐ False
- 9) `print_val(9.0)`
☐ True
☐ False



©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Mathematical functions

A function is commonly defined to compute a mathematical operation involving several numerical parameters and returning a numerical result. Ex: The following program uses a function to convert a person's height in U.S. units (feet and inches) into total centimeters.

PARTICIPATION ACTIVITY

6.3.4: Program with a function to convert height in feet/inches to centimeters.



```
1 CM_PER_INCH = 2.54
2 INCHES_PER_FOOT = 12
3
4 def height_US_to_cm(feet, inches):
5     """Converts height in feet/inches to centimeters"""
6     total_inches = feet * INCHES_PER_FOOT + inches
7     cm = total_inches * CM_PER_INCH
8     return cm
9
10 feet = 6
11 inches = 4
12
13 centimeters = height_US_to_cm(feet, inches)
14 print('Centimeters:', centimeters)
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

<< First < Back Step 1 of 12 Forward > Last >

→ line that has just executed

→ next line to execute

[Sidenotes: Most Americans only know their height in feet/inches, not in total inches. Human average height is increasing, attributed largely to better nutrition. ([Wikipedia: Human height](#))]

zyDE 6.3.1: Temperature conversion.

Complete the program by writing and calling a function that converts a temperature from Celsius into Fahrenheit. Use the formula $F = C \times 9/5 + 32$. Test your program knowing 50 Celsius is 122 Fahrenheit.

[Load default template](#)

```

1 def c_to_f():
2     # FIXME
3     return # FIXME: Finish
4
5 temp_c = float(input('Enter temperature in Celsius: '))
6 temp_f = None
7
8 # FIXME: Call conversion function
9 # temp_f = ??
10
11 # FIXME: Print result
12 # print('Fahrenheit:', temp_f)
13

```

50

Run

Hierarchical function calls

A function's statements may include function calls, known as **hierarchical function calls** or **nested function calls**. Code such as `user_input = int(input())` consists of such a hierarchical function call, wherein the `input()` function is called and evaluates to a value that is then passed as an argument to the `int()` function.

PARTICIPATION ACTIVITY

6.3.5: Hierarchical function calls

```

1 def calc_circle_area(circle_diameter):
2     pi_val = 3.14159265
3
4     circle_radius = circle_diameter / 2.0
5     circle_area = pi_val * circle_radius * circle_radius
6     return circle_area
7
8 def pizza_calories(pizza_diameter):
9     calories_per_square_inch = 16.7    # Regular crust pepperoni p:
10
11     total_calories = calc_circle_area(pizza_diameter) * calories_p

```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

```

12     return total_calories
13
14
15 print('12 inch pizza has {:.2f} calories.'.format(pizza_calories(12)))
16 print('14 inch pizza has {:.2f} calories.'.format(pizza_calories(14)))

```

<< First < Back Step 1 of 26 Forward > Last >>

→ line that has just executed

→ next line to execute

Frames

Objects

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

PARTICIPATION ACTIVITY

6.3.6: Hierarchical function calls.

Complete the `pizza_calories_per_slice()` function to compute the calories for a single slice of pizza. A `pizza_calories()` function returns a pizza's total calories given the pizza diameter passed as an argument. A `pizza_slices()` function returns the number of slices in a pizza given the pizza diameter passed as an argument.

```

def pizza_calories_per_slice(pizza_diameter):
    total_calories = <placeholder_A>
    calories_per_slice = <placeholder_B>
    return calories_per_slice

```

- 1) Type the expression for `placeholder_A` to compute the total calories for a pizza with diameter `pizza_diameter`.

`total_calories =`

Check

Show answer

- 2) Type the expression for `placeholder_B` to compute the calories per slice.

`calories_per_slice =`

Check

Show answer

CHALLENGE ACTIVITY

6.3.1: Enter the output of the returned value.

Start

Type the program's output.

```

def change_value(x):
    return x + 2
print(change_value(2))

```


2

Check

Next

View solution ▼ (Instructors only)

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

**CHALLENGE
ACTIVITY**

6.3.2: Function call in expression.



Assign `max_sum` with the greater of `num_a` and `num_b`, PLUS the greater of `num_y` and `num_z`.
Use just one statement. Hint: Call `find_max()` twice in an expression.

Sample output with inputs: 5.0 10.0 3.0 7.0

max_sum is: 17.0

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

```
1 def find_max(num_1, num_2):
2     max_val = 0.0
3
4     if (num_1 > num_2): # if num1 is greater than num2,
5         max_val = num_1 # then num1 is the maxVal.
6     else: # Otherwise,
7         max_val = num_2 # num2 is the maxVal
8     return max_val
9
10 max_sum = 0.0
11
12 num_a = float(input())
13 num_b = float(input())
14 num_y = float(input())
15 num_z = float(input())
16
17 ''' Your solution goes here '''
18
19 print('max_sum is:', max_sum)
```

Run

View solution ▼ (Instructors only)

↓ **Download student submissions** ⓘ

**CHALLENGE
ACTIVITY**

6.3.3: Function definition: Volume of a pyramid.



Define a function `pyramid_volume` with parameters `base_length`, `base_width`, and `pyramid_height`, that returns the volume of a pyramid with a rectangular base.

Sample output with inputs: 4.5 2.1 3.0

Volume for 4.5, 2.1, 3.0 is: 9.45

Relevant geometry equations:

Volume = base area x height x 1/3

Base area = base length x base width.

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Run

View solution  (Instructors only)

 Download student submissions


<

03/05/20 10:28 591419

Alexey Munishkin

UCSCCSE20NawabWinter2020

6.4 Dynamic typing

Dynamic and static typing

A programmer can pass any type of object as an argument to a function. Consider a function `add(x, y)` that adds the two parameters:

A programmer can call the `add()` function using two integer arguments, as in `add(5, 7)`, which returns a value of 12. Alternatively, a programmer can pass in two string arguments, as in `add('Tora', 'Bora')`, which would concatenate the two strings and return `'ToraBora'`.

PARTICIPATION ACTIVITY

6.4.1: Polymorphic functions.

1

def add(x, y):

2

return x + y

3

4

print('add(5, 7) is', add(5, 7))

5

print("add('Tora', 'Bora') is", add('Tora', 'Bora'))

<< First

< Back

Step 1 of 9

Forward >

Last >>

→ line that has just executed

→ next line to execute

Frames

Objects

The function's behavior of being able to add together different types is a concept called **polymorphism**. Polymorphism is an inherent part of the Python language. For example, consider the multiplication operator `*`. If the two operands are numbers, then the result is the product of those two numbers. If one operand is a string and the other an integer (e.g., `'x' * 5`) then the result is a repetition of the string 5 times: `'xxxxx'`.

Python uses **dynamic typing** to determine the type of objects as a program executes. Ex: The consecutive statements `num = 5` and `num = '7'` first assign `num` to an integer type, and then a string type. The type of `num` can change, depending on the value it references. The interpreter is responsible for checking that all operations are valid as the program executes. If the function call `add(5, '100')` is evaluated, an error is generated when adding the string to an integer.

In contrast to dynamic typing, many other languages like C, C++, and Java use **static typing**, which requires the programmer to define the type of every variable and every function parameter in a program's source code. Ex: `string name = "John"` would declare a string variable in C and C++. When the source code is compiled, the compiler attempts to detect non type-safe operations, and halts the compilation process if such an operation is found.

Dynamic typing typically allows for more flexibility in terms of the code that a programmer can write, but at the expense of potentially introducing more bugs, since there is no compilation process by which types can be checked.¹

- 1) Polymorphism refers to how an operation depends on the involved object types.
☐ True
☐ False
- 2) A programmer can pass only string arguments to a user-defined function.
☐ True
☐ False
- 3) Static-typed languages require that the type of every variable is defined in the source code.
☐ True
☐ False
- 4) A dynamic-typed language like Python checks that an operation is valid when that operation is executed by the interpreter. If the operation is invalid, a run-time error occurs.
☐ True
☐ False

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

(*1) Python uses **duck typing**, a form of dynamic typing based on the maxim "If a bird walks, swims, and quacks like a duck, then call it a duck." For example, if an object can be concatenated, sliced, indexed, and converted to ASCII, doing everything that a string can do, then treat the object like a string.

6.5 Reasons for defining functions

Improving program readability

Programs can become hard for humans to read and understand. Decomposing a program into functions can greatly aid program readability, helping yield an initially correct program, and easing future maintenance. The following program contains two user-defined functions, making the main program (after the function definitions) easier to read and understand. For larger programs, the effect is even greater.

Figure 6.5.1: With user-defined functions, the main program is easy to understand.

<div>Enter number of steps walked: 1000 Feet: 3000 Calories: 50</div>

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

```
def steps_to_feet(num_steps):
    feet_per_step = 3
    feet = num_steps * feet_per_step
    return feet

def steps_to_calories(num_steps):
    steps_per_minute = 70.0
    calories_per_minute_walking = 3.5

    minutes = num_steps / steps_per_minute
    calories = minutes * calories_per_minute_walking
    return calories

steps = int(input('Enter number of steps walked: '))
feet = steps_to_feet(steps)
print('Feet:', feet)

calories = steps_to_calories(steps)
print('Calories:', calories)
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Figure 6.5.2: Without user-defined functions, the main program is harder to read and understand.

```
feet_per_step = 3
steps_per_minute = 70.0
calories_per_minute_walking = 3.5

steps = int(input('Enter number of steps walked: '))

feet = steps * feet_per_step
print('Feet:', feet)

minutes = steps / steps_per_minute
calories = minutes * calories_per_minute_walking
print('Calories:', calories)
```

```
Enter number of steps walked: 1000
Feet: 3000
Calories: 50
```

PARTICIPATION ACTIVITY

6.5.1: Improved readability.

Consider the above examples.

- 1) In the example *without* functions, how many statements are in the main program?
 - ☐ 5
 - ☐ 9
- 2) In the example *with* functions, how many statements are in the main program?
 - ☐ 5
 - ☐ 9
- 3) Which has fewer *total* lines of code (including blank lines), the program with or without functions?
 - ☐ With
 - ☐ Without
 - ☐ Same

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Modular program development

Programmers commonly use functions to write programs modularly. **Modular development** is the process of dividing a program into separate modules that can be developed and tested separately and then integrated into a single program.

A programmer can use function stubs (described in depth elsewhere) to capture the high-level behavior of the required functions (or modules) before diving into details of each function, like planning a route for a road trip before starting to drive.

Avoid writing redundant code

A function can be defined once, then called from multiple places in a program, thus avoiding redundant code. Examples of such functions are math module functions like `sqrt()` that relieve a programmer from having to write several lines of code each time a square root needs to be computed.

The skill of decomposing a program's behavior into a good set of functions is a fundamental part of programming that helps characterize a good programmer. Each function should have easily-recognizable behavior, and the behavior of the main program (and any function that calls other functions) should be easily understandable via the sequence of function calls.

A general guideline (especially for beginner programmers) is that a function's definition usually shouldn't have more than about 30 lines of code, although this guideline is not a strict rule.

PARTICIPATION ACTIVITY

6.5.2: Redundant code can be replaced by multiple calls to one function.



Animation content:

undefined

Animation captions:

1. Circle area is calculated twice, leading to redundant code.
2. The redundant code can be replaced by defining a `calc_circle_area()` function.
3. Then main program is simplified by calling the `calc_circle_area()` function from multiple places in the program.

PARTICIPATION ACTIVITY

6.5.3: Reasons for defining functions.



1) A key reason for creating functions is to help the program run faster.

- ☐ True
☐ False

2) Avoiding redundancy means to avoid calling a function from multiple places in a program.

- ☐ True
☐ False

3) If a function's internal statements are revised, all function calls will have to be modified too.

- ☐ True
☐ False

4) A benefit of functions is to increase redundant code.

- ☐ True
☐ False

Write a function so that the main program below can be replaced by the simpler code that calls function `mph_and_minutes_to_miles()`. Original main program:

```
miles_per_hour = float(input())
minutes_traveled = float(input())
hours_traveled = minutes_traveled / 60.0
miles_traveled = hours_traveled * miles_per_hour

print('Miles: {:.f}'.format(miles_traveled))
```

Sample output with inputs: 70.0 100.0

Miles: 116.666667

```
1
2 ''' Your solution goes here '''
3
4 miles_per_hour = float(input())
5 minutes_traveled = float(input())
6
7 print('Miles: {:.f}'.format(mph_and_minutes_to_miles(miles_per_hour, minutes_traveled)))
```

Run

View solution ▼ (Instructors only)

↓ Download student submissions ⓘ

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

6.6 Function stubs

Incremental development and function stubs

Programs are typically written using **incremental development**, meaning a small amount of code is written and tested, then a small amount more (an incremental amount) is written and tested, and so on.

To assist with the incremental development process, programmers commonly introduce **function stubs**, which are function definitions whose statements haven't been written yet. The benefit of a function stub is that the high-level behavior of the program can be captured before diving into details of each function, akin to planning the route of a road trip before starting to drive. Capturing high-level behavior first may lead to better-organized code, reduced development time, and even to code with fewer bugs.

A programmer writing a function stub should consider whether or not calling the unwritten function is a valid operation. Simply doing nothing and returning nothing may be acceptable early in the development of a larger program. One approach is to use the **pass** keyword, which performs no operation except to act as a placeholder for a required statement.

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Figure 6.6.1: Using the pass statement in a function stub performs no operation.

<pre>def steps_to_feet(num_steps): feet_per_step = 3 feet = num_steps * feet_per_step return feet def steps_to_calories(num_steps): pass steps = int(input('Enter number of steps walked: ')) feet = steps_to_feet(steps) print('Feet:', feet) calories = steps_to_calories(steps) print('Calories:', calories)</pre>	<pre>Enter number of steps walked: 1000 Feet: 3000 Calories: None ... Enter number of steps walked: 0 Feet: 0 Calories: None ... Enter number of steps walked: 99999 Feet: 299997 Calories: None</pre>
---	--

The program above has a function stub in place of the `steps_to_calories()` function. The function contains a single `pass` statement because at least one statement is required in any user-defined function.

Another useful approach is to print a message when a function stub is called, thus alerting the user to the missing function statements. Good practice is for a stub to return -1 for a function that will have a return value. The following function stub could be used to replace the `steps_to_calories()` stub in the program above:

Figure 6.6.2: A function stub using a print statement.

```
def steps_to_calories(steps):
    print('FIXME: finish steps_to_calories')
    return -1
```

In some cases, a programmer may want a program to stop executing if an unfinished function is called. Ex: A program that requires user input should not execute if the user-defined function that gets input is not completed. In such cases, a **NotImplementedError** can be generated with the statement `raise NotImplementedError`. The `NotImplementedError` indicates that the function is not implemented and causes the program to stop execution. `NotImplementedError` and the "raise" keyword are explored elsewhere in material focusing on *exceptions*. The following demonstrates an error being generated by a function stub:

Figure 6.6.3: Stopping the program using `NotImplementedError` in a function stub.

<pre>import math def get_points(num_points): """Get num_points from the user. Return a list of (x,y) tuples.""" raise NotImplementedError def side_length(p1, p2): return math.sqrt((p2[0] - p1[0])**2 + (p2[1] - p1[1])**2) def get_perimeter_length(points): perimeter = side_length(points[0], points[1]) perimeter += side_length(points[0], points[2]) perimeter += side_length(points[1], points[2]) return perimeter coordinates = get_points(3) print('Perimeter of triangle:', get_perimeter_length(coordinates))</pre>	<pre>Traceback (most recent call last): File "<stdin>", line 10, in get_perimeter_length File "<stdin>", line 2, in get_points NotImplementedError</pre>
--	--

**PARTICIPATION
ACTIVITY**

6.6.1: Incremental development and function stubs.

1) Incremental development may involve more frequent testing, but ultimately leads to faster development of a program.

- ☐ True
☐ False

2) The main advantage of function stubs is that they ultimately lead to faster-running programs.

- ☐ True
☐ False

3) A pass statement should be used in a function stub when the programmer wants the stub to stop program execution when called.

- ☐ True
☐ False

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

**CHALLENGE
ACTIVITY**

6.6.1: Function stubs: Statistics.

Define stubs for the functions `get_user_num()` and `compute_avg()`. Each stub should print "FIXME: Finish function_name()" followed by a newline, and should return -1. Each stub must also contain the function's parameters.

Sample output with two calls to `get_user_num()` and one call to `compute_avg()`:

```
FIXME: Finish get_user_num()
FIXME: Finish get_user_num()
FIXME: Finish compute_avg()
Avg: -1
```

```
1
2 ''' Your solution goes here '''
3
4 user_num1 = 0
5 user_num2 = 0
6 avg_result = 0
7
8 user_num1 = get_user_num()
9 user_num2 = get_user_num()
10 avg_result = compute_avg(user_num1, user_num2)
11
12 print('Avg:', avg_result)
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Run

View solution ▼ (Instructors only)

6.7 Functions with branches/loops

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Example: ebay_fee() function

Note: This section requires knowledge of if-else and loop statements.

A function's block of statements may include branches, loops, and other statements. The following example uses a function to compute the amount that an online auction/sales website charges a customer who sells an item online.

Figure 6.7.1: Function example: Determining fees given an item selling price for an auction website.

```
def ebay_fee(selling_price):
    """Returns the fees charged by ebay.com given the
    selling
    price of fixed-price books, movies, music, or video
    games.
    fee is $0.50 to list plus 13% of selling price up
    to $50.00,
    5% of amount from $50.01 to $1000.00, and
    2% for amount $1000.01 or more."""

    p50 = 0.13 # for amount $50 and lower
    p50_to_1000 = 0.05 # for $50.01-$1000
    p1000 = 0.02 # for $1000.01 and higher
    fee = 0.50 # fee to list item

    if selling_price <= 50:
        fee = fee + (selling_price*p50)
    elif selling_price <= 1000:
        fee = fee + (50*p50) + ((selling_price-
50)*p50_to_1000)
    else:
        fee = fee + (50*p50) + ((1000-50)*p50_to_1000)
        + ((selling_price-1000)*p1000)

    return fee

selling_price = float(input('Enter item selling price
(ex: 65.00): '))
print('Ebay fee: $', ebay_fee(selling_price))
```

```
Enter item selling price (ex:
65.00): 9.95
eBay fee: $ 1.7934999999999999
...
Enter item selling price (ex:
65.00): 40
eBay fee: $ 5.7
...
Enter item selling price (ex:
65.00): 100
eBay fee: $ 9.5
...
Enter item selling price (ex:
65.00): 500
eBay fee: $ 29.5
...
Enter item selling price (ex:
65.00): 2000
eBay fee: $ 74.5
```

PARTICIPATION ACTIVITY

6.7.1: Analyzing the ebay fee function.

- 1) For any call to ebay_fee(), how many assignment statements will execute?

Check Show answer

- 2) What does ebay_fee() return if its argument is 0.0 (show your answer in the form #.#)?

Check Show answer

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

- 3) What does `ebay_fee()` return if its argument is 100.0 (show your answer in the form #.#)?

Check [Show answer](#)

- 4) Write a function call using the `ebay_fee()` function to determine the fee for a selling price of 15.23, storing the result in a variable named `my_fee`.

Check [Show answer](#)

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Example: Numbers program with multiple functions

The following is a more complex example with user-defined functions. Notice that functions keep the program's behavior readable and understandable.

zyDE 6.7.1: User-defined functions make a program easier to understand.

The problem below uses the function `get_numbers()` to read a number of integers from the user. Three unfinished functions are defined, which should print only certain types of numbers that the user entered. Complete the unfinished functions, adding loops and branches where necessary. Match the output with the below sample:

```
Enter 5 integers:
0 5
1 99
2 -44
3 0
4 12
Numbers: 5 99 -44 0 12
Odd numbers: 5 99
Negative numbers: -44
```

[Load default template...](#)

5 99 -44 0 12

Run

```
1 size = 5
2
3 def get_numbers(num):
4     numbers = []
5     user_input = input('Enter {} integers: ')
6
7     i = 0
8     for token in user_input.split():
9         number = int(token) # Convert to int
10        numbers.append(number) # Add to numbers
11
12        print(i, number)
13        i += 1
14
15    return numbers
16
17 def print_all_numbers(numbers):
18     # Print numbers
19     print('Numbers:')
20
21
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

PARTICIPATION ACTIVITY

6.7.2: Analyzing the numbers program.

- 1) For a single execution of the program, how many calls to user-defined functions are made?

[Check](#)[Show answer](#)**CHALLENGE
ACTIVITY**

6.7.1: Function with branch: Popcorn.



Define function `print_popcorn_time()` with parameter `bag_ounces`. If `bag_ounces` is less than 3, print "Too small". If greater than 10, print "Too large". Otherwise, compute and print $6 * \text{bag_ounces}$ followed by "seconds". End with a newline.

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Sample output with input: 7

42 seconds

```
1
2 ''' Your solution goes here '''
3
4 user_ounces = int(input())
5 print_popcorn_time(user_ounces)
```

Run

View solution (Instructors only)

[Download student submissions](#)

**CHALLENGE
ACTIVITY**

6.7.2: Function with loop: Shampoo.



Write a function `shampoo_instructions()` with parameter `num_cycles`. If `num_cycles` is less than 1, print "Too few.". If more than 4, print "Too many.". Else, print "N : Lather and rinse." `num_cycles` times, where N is the cycle number, followed by "Done.".

Sample output with input: 2

```
1 : Lather and rinse.
2 : Lather and rinse.
Done.
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Hint: Define and use a loop variable.

Run

View solution ▼ *(Instructors only)*

[Download student submissions](#)
ⓘ

©zyBooks 03/05/20 10:28 591419

Alexey Munishkin

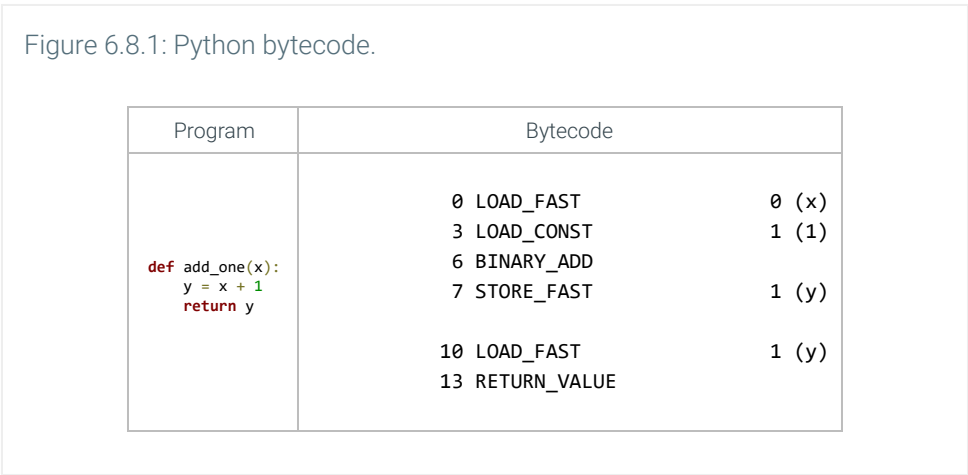
UCSCCSE20NawabWinter2020

6.8 Functions are objects

Functions as objects

A function is also an object in Python, having a type, identity, and value. A function definition like `def print_face():` creates a new function object with the name `print_face` bound to that object.

A part of the value of a function object is compiled **bytecode** that represents the statements to be executed by the function. A bytecode is a low-level operation, such as adding, subtracting, or loading from memory. One Python statement might require multiple bytecodes operations. Ex: The function below adds 1 to an argument and returns the result. The corresponding bytecode for the function requires 4 bytecode operations to perform the addition, and 2 to return the result.



All Python code is compiled before being executed by the interpreter. Statements entered in an interactive interpreter are compiled immediately, then executed. Modules are compiled when imported, and functions are compiled when the interpreter evaluates the function definition.

A statement like `print_face()` causes the function object to execute a call operation, which in turn executes the function's bytecode. A programmer never has to deal with bytecode – bytecode is used internally by the interpreter.

Because a function is an object, a function can be used in an assignment statement just like other objects. This is illustrated in the following animation.

Animation content:

undefined

Animation captions:

1. `def print_face()` creates a new function object.
2. The compiled bytecode of `print_face` function is stored in the function object.
3. When `print_face()` is called, the `print_face()` function runs.
4. Calling `func()` is the same as calling `print_face()`.

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

The interpreter creates a new function object when the definition `def print_face()` is evaluated. The function object contains as part of its value the function's bytecode. Since a function is just an object, assignment operations work the same: `func = print_face` binds the name `func` to the same object as `print_face`, thus creating multiple names for a single function. Both `func()` and `print_face()` perform the same call operation and jump execution to `print_face`.

Functions can be passed like any other object as an argument to another function. Consider the following example, which defines two different functions `human_head()` and `monkey_head()`. A third function `print_figure()` accepts a function as an argument, calling that function to print a head, and then printing a body.

Figure 6.8.2: Functions can be passed as arguments.

```
def human_head():
    print(' |||| ')
    print('  o  o')
    print('   > ')
    print('  ooooo')
    return

def monkey_head():
    print(' .-.-. ')
    print(' / .-.-. \')
    print(' ( o o ) ')
    print(' /  "  \')
    print(' \ .-.-. /')
    print(' /"""\')
    return

def print_figure(face):
    face() # Print the face
    print(' | ')
    print(' --|-- ')
    print(' / | \')
    print('@ | @')
    print(' | ')
    print(' /|\')
    print(' @ | @')
    return

choice = int(input('Enter "1" to draw monkey, "2" for
human: '))

if choice == 1:
    print_figure(monkey_head)
elif choice == 2:
    print_figure(human_head)
```

```
Enter "1" to draw monkey, "2" for
human: 1
.-.-.
/ .-.-. \
( o o )
/  "  \
 \ .-.-. /
 /"""\
|
--|--
/ | \
@ | @
|
/|\
@ | @
...
Enter "1" to draw monkey, "2" for
human: 2
||||
 o  o
  >
 ooooo
 |
--|--
/ | \
@ | @
|
/|\
@ | @
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Passing functions as arguments can sometimes improve the readability of code. The above example could have been implemented using an if statement to call either `human_head()` or `monkey_head()` followed by a call to a `print_body()` function. However, the code is simplified by reducing the required number of function calls in the first code block to the more simple `print_figure(face)`.

Whereas objects like integers support many operations (adding, subtracting, etc.), functions really only support the call operation.¹



1) Functions are compiled into bytecode when the function definition is evaluated by the interpreter.

- ☐ True
☐ False

2) The output of the following program is

```
def cat():  
    print('meow')  
  
'meow': def pig():  
         print('oink')  
  
        cat = pig  
        cat()
```

- ☐ True
☐ False

3) If `my_func1()` and `my_func2()` are defined functions, then the expression `my_func1 + my_func2` returns a valid value.

- ☐ True
☐ False

4) The expression `my_func1(my_func2())` passes the `my_func2` function object as an argument to `my_func1`.

- ☐ True
☐ False

(*1) Functions also support adding attributes with the attribute reference `."` operator, but that concept is out of scope for the discussion here.

6.9 Functions: Common errors

Copy-paste errors

A [common error](#) is to copy-and-paste code among functions but then not complete all necessary modifications to the pasted code. For example, a programmer might have developed and tested a function to convert a temperature value in Celsius to Fahrenheit, and then copied and modified the original function into a new function to convert Fahrenheit to Celsius as shown:

Figure 6.9.1: Copy-paste common error: Pasted code not properly modified. Find error on the right.

```
def celsius_to_fahrenheit(celsius):  
    temperature = (9.0/5.0) * celsius  
    fahrenheit = temperature + 32  
  
    return fahrenheit
```

```
def fahrenheit_to_celsius(fahrenheit):  
    temperature = fahrenheit- 32  
    celsius = temperature * (5.0/9.0)  
  
    return fahrenheit
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

The programmer forgot to change the return statement to return **celsius** rather than **fahrenheit**. Copying-and-pasting code is a common and useful time-saver and can reduce errors by starting with known-correct code. Our advice is that when you copy-paste code, be extremely vigilant in making all necessary modifications. Just as the awareness that dark alleys or wet roads may be dangerous can cause you to vigilantly observe your surroundings or drive carefully, the awareness that copying-and-pasting is a common source of errors may cause you to more vigilantly ensure you modify a pasted function correctly.

**PARTICIPATION
ACTIVITY**

6.9.1: Copy-pasted sum-of-squares code.

Original parameters were `num1`, `num2`, `num3`.
Original code was:

```
sum = (num1 * num1) + (num2 * num2) + (num3 * num3)
return sum
```


New parameters are `num1`, `num2`, `num3`, `num4`. Find the error in the copy-pasted new code below.

1)

```
sum = (num1 * num1) + (num2 * num2) +
(num3 * num3) + (num3 * num4)
return sum
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Return errors

Another common error is to return the wrong variable, like if `return temperature` had been used in the temperature conversion program by accident. The function will work and sometimes even return the correct value.

Another common error is to fail to return a value for a function. If execution reaches the end of a function's statements without encountering a return statement, then the function returns a value of **None**. If the function is expected to return an actual value, then such an assignment can cause confusion.

**PARTICIPATION
ACTIVITY**

6.9.2: Missing return common error.

```
1 def steps_to_feet(num_steps):
2     feet_per_step = 3
3     feet = num_steps * feet_per_step
4     # Missing return statement!
5
6     feet_per_mile = 5280
7     steps = 1000
8
9     feet = steps_to_feet(steps)
10    print("Distance walked in feet:", feet)
```

<< First < Back Step 1 of 9 Forward > Last >>

→ line that has just executed
→ next line to execute

Frames Objects

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

The program above produces unexpected output, leading to a bug that's hard to find. The program does not contain syntax errors, but does contain a logic error because the function `steps_to_feet()` always returns a value **None**.

**PARTICIPATION
ACTIVITY**

6.9.3: Common function errors.

< >

1) Forgetting to return a value from a function is a common error.

- ☐ True
☐ False

2) Copying-and-pasting code can lead to common errors if all necessary changes are not made to the pasted code.

- ☐ True
☐ False

3) Returning the incorrect variable from a function is a common error.

- ☐ True
☐ False

4) Is this function correct for squaring an integer?

```
def sqr (a):  
    t = a * a
```

- ☐ Yes
☐ No

5) Is this function correct for squaring an integer?

```
def sqr (a):  
    t = a * a  
    return a
```

- ☐ Yes
☐ No

CHALLENGE ACTIVITY

6.9.1: Function errors: Copying one function to create another.

Using the `celsius_to_kelvin` function as a guide, create a new function, changing the name to `kelvin_to_celsius`, and modifying the function accordingly.

Sample output with input: 283.15

```
10.0 C is 283.15 K  
283.15 K is 10.0 C
```

```
1 def celsius_to_kelvin(value_celsius):  
2     value_kelvin = 0.0  
3  
4     value_kelvin = value_celsius + 273.15  
5     return value_kelvin  
6  
7 ''' Your solution goes here '''  
8  
9 value_c = 10.0  
10 print(value_c, 'C is', celsius_to_kelvin(value_c), 'K')  
11  
12 value_k = float(input())  
13 print(value_k, 'K is', kelvin_to_celsius(value_k), 'C')
```

Run

View solution ▼ *(Instructors only)*

↓ Download student submissions i

< >

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

6.10 Scope of variables and functions

Variable and function scope

A variable or function object is only visible to part of a program, known as the object's **scope**. When a variable is created inside a function, the variable's scope is limited to *inside* that function. In fact, because a variable's name does not exist until bound to an object, the variable's scope is actually limited to *after* the first assignment of the variable until the end of the function. The following program highlights the scope of variable `total_inches`.

Figure 6.10.1: Variable scope.

```
centimeters_per_inch = 2.54
inches_per_foot = 12

def height_US_to_centimeters(feet, inches):
    """ Converts a height in feet/inches to centimeters."""
    total_inches = (feet * inches_per_foot) + inches # Total inches
    centimeters = total_inches * centimeters_per_inch
    return centimeters

feet = int(input('Enter feet: '))
inches = int(input('Enter inches: '))

print('Centimeters:', height_US_to_centimeters(feet, inches))
```

Local variable scope extends from assignment to end of function. Global variable scope extends to end of file.

The function's variables `total_inches` and `centimeters` are invisible to the code outside of the function and cannot be used. Such variables defined inside a function are called **local variables**.

Global variables

In contrast, a variable defined outside of a function is called a **global variable**. A global variable's scope extends from assignment to the end of the file and can be accessed inside of functions.

A **global** statement must be used to *change* the value of a global variable inside of a function. The following shows two programs: the right uses a global statement to allow the modification of global variable `employee_name` inside of the `get_name` function.

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Figure 6.10.2: The global statement (right) allows modifying a global variable.



<pre> employee_name = 'N/A' def get_name(): name = input('Enter employee name:') employee_name = name get_name() print('Employee name:', employee_name) </pre>	<pre> employee_name = 'N/A' def get_name(): global employee_name name = input('Enter employee name:') employee_name = name get_name() print('Employee name:', employee_name) </pre>
<div>Enter employee name: Romeo Montague</div> <div>Employee name: N/A</div>	<div>Enter employee name: Juliet Capulet</div> <div>Employee name: Juliet Capulet</div>

©zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020





The global statement must be applied to any global variable that is to be assigned to in a function. Modification of mutable global variables, such as list or dict containers, does not require a global statement if a programmer is adding or removing elements from the container. The reasons for requiring a global statement are discussed in more detail later.

Assignment of global variables in functions should be used sparingly. If a local variable (including a parameter) has the same name as a global variable, then the name refers to the local item (so the global item is inaccessible inside such a function). Such naming can be very confusing to a reader. Furthermore, if a function updates the global variable, then that function's behavior is no longer limited to its parameters and return value; the function may have *side effects* that are hard for a programmer to recognize. Good practice is to limit the use of global variables to defining constants that are independent of any function. Global variables should generally be avoided (with a few exceptions), especially by beginner programmers.

A function also has scope, which extends from the function's definition to the end of the file. To be able to call a function, the interpreter must have already evaluated the function definition (thus binding the function name to a function object). An attempt to call a function before a function has been defined results in an error.

Figure 6.10.3: Function definitions must be evaluated before that function is called.

<pre> employee_name = 'N/A' get_name() print('Employee name:', employee_name) def get_name(): global employee_name name = input('Enter employee name:') employee_name = name </pre>	<div>NameError: name 'get_name' is not defined</div>
---	--

PARTICIPATION ACTIVITY	6.10.1: Variable/ function scope.	
1) A local variable is defined inside a function, while a global variable is defined outside any function.		
<div> <input type="radio"/> True <input type="radio"/> False </div>		
2) A local variable's scope extends from a function definition's ending colon ":" to the end of the function.		
<div> <input type="radio"/> True <input type="radio"/> False </div>		
3)		

A global statement must be used whenever a global variable is to be read or written in a function.

- ☐ True
☐ False

4) A function definition must be evaluated by the interpreter before the function can be called.

- ☐ True
☐ False

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

6.11 Namespaces and scope resolution

Namespace

A **namespace** maps names to objects. The Python interpreter uses namespaces to track all of the objects in a program. For example, when executing `z = x + y`, the interpreter looks in a namespace to find the value of the objects referenced by `x` and `y`, evaluates the expression, and then updates `z` in the namespace with the expression's result.

PARTICIPATION ACTIVITY

6.11.1: Namespaces.

Animation content:

undefined

Animation captions:

1. Global variables are tracked in the global namespace.
2. The value of variables is found by looking in the namespace.

In fact, a namespace is actually just a normal Python dictionary whose keys are the names and whose values are the objects. A programmer can examine the names in the current local and global namespace by using the `locals()` and `globals()` built-in functions.

Figure 6.11.1: Using the `globals()` to get namespace names.

```
print('Initial global  
namespace: ')  
print(globals())  
  
my_var = "This is a variable"  
print('\nCreated new  
variable')  
print(globals())  
  
def my_func():  
    pass  
  
print('\nCreated new  
function')  
print(globals())
```

```
Initial global namespace:  
{}  
  
Created new variable  
{'my_var': 'This is a variable'}  
  
Created new function  
{'my_func': <function my_func at 0x2349d4>, 'my_var': 'This  
is a variable'}
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

By default, a few names already exist in the global namespace – those names have been omitted in the output for brevity. Notice that `my_var` and `my_func` are added into the namespace once assigned.

Scope and scope resolution

Scope is the area of code where a name is visible. Namespaces are used to make scope work. Each scope, such as global scope or a local function scope, has its own namespace. If a namespace contains a name at a specific location in the code, then that name is visible and a programmer can use it in an expression.

There are at least three nested scopes that are active at any point in a program's execution: ¹

- 1. Built-in scope – Contains all of the built-in names of Python, such as `int()`, `str()`, `list()`, `range()`, etc.
- 2. Global scope – Contains all globally defined names outside of any functions.
- 3. Local scope – Usually refers to scope within the currently executing function, but is the same as global scope if no function is executing.

When a name is referenced in code, the local scope's namespace is the first checked, followed by the global scope, and finally the built-in scope. If the name cannot be found in any namespace, the interpreter generates a `NameError`. The process of searching for a name in the available namespaces is called **scope resolution**.

PARTICIPATION
ACTIVITY

6.11.2: Scope resolution.

Animation content:

undefined

Animation captions:

- 1. Global variables are added to the global namespace.
- 2. Calling a function creates a new local namespace for local variables.
- 3. Variables are resolved by looking in the local namespace first, then global, then built-in.
- 4. The local namespace is removed when the function returns.

As the code executes, the global scope namespace is filled with names as they are defined. The function call creates a new namespace to track the variables in the function's local scope. The new local namespace automatically contains the parameter value `cals_left`. When the expression `cals_left - soda_cals` is evaluated, the interpreter finds `cals_left` in the local namespace, then finds `soda_cals` in the global namespace after unsuccessfully searching the local namespace.

PARTICIPATION
ACTIVITY

6.11.3: Namespaces and scopes.

Namespace	locals()	Scope resolution	Scope
		Maps the visible names in a scope to objects.	
		The area of code where a name is visible.	
		Returns a dictionary of the names found in the local namespace.	
		The process of searching namespaces for a name.	
			Reset



Given the following program, select the namespace that each name would belong to.

```
import random

player_name = 'Gandalf'
player_type = 'Wizard'

def roll():
    """Returns a roll of a 20-sided die"""
    number = random.randint(1, 20)
    return number

print('A troll attacks!')
troll_roll = roll()
player_roll = roll()

print('Player: {} Troll: {}'.format(str(player_roll), str(troll_roll)))
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

1) player_name

- ☐ local
☐ global
☐ built-in



2) roll

- ☐ local
☐ global
☐ built-in



3) number

- ☐ local
☐ global
☐ built-in



4) str

- ☐ local
☐ global
☐ built-in



More scoping and namespaces

The concept of scopes and namespaces helps to explain how multiple variables can share the same name, yet have different values. Consider the following program that first creates a variable `tmp` in the global namespace, then creates another variable named `tmp` in a local function. The assignment statement in the `avg()` function creates a new variable within the function's local namespace. When the function returns, the namespace is deleted as well (since the local variables are now out of scope). The later statement `print 'Sum: {:.f}'.format(tmp)` looks up the name `tmp` in the global scope, finding the `tmp` previously created with the statement `tmp = a + b`.

Note that the Python Tutor tool below uses the term "frame" in place of "namespace".

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020



```
1 def avg(a, b):
2     tmp = (a + b) / 2.0 # Creates tmp in local namespace
3     return tmp
4
5 a = 5
6 b = 10
7 tmp = a + b # Creates tmp in global namespace
```



```

8
9 print('Avg: {:.f}'.format(avg(a, b)))
10 print('Sum: {:.f}'.format(tmp))

```

<< First < Back Step 1 of 10 Forward > Last >>

→ line that has just executed

→ next line to execute

Frames

Objects

©zyBooks 03/05/20 10:28 591419

Alexey Munishkin

UCSCCSE20NawabWinter2020

By default, any assignment statement automatically creates (or modifies) a name in the local namespace only, even if the same name exists in a higher global or built-in scope. A global statement such as **global tmp** forces the interpreter to consider the variable in the global scope, thus allowing modification of existing global variables instead of creating local variables.

PARTICIPATION ACTIVITY

6.11.6: Namespace and scope.

- 1) A namespace is how the Python interpreter restricts variables to a specific scope.
 - ☐ True
 - ☐ False
- 2) Whenever a function is called, a local namespace is created for that function.
 - ☐ True
 - ☐ False
- 3) The same name cannot be in multiple namespaces.
 - ☐ True
 - ☐ False
- 4) If a programmer defines a function called **list()**, the program will crash because there is already a built-in function with the same name.
 - ☐ True
 - ☐ False

(*1) Actually, there are four levels of scopes. We have omitted a level between the local function scope and global scope for clarity. It is possible to define a function within another function – in such a case the scope of the outer function is checked before the global scope is checked.

©zyBooks 03/05/20 10:28 591419

Alexey Munishkin

UCSCCSE20NawabWinter2020

6.12 Function arguments

Function arguments and mutability

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.

Animation content:

undefined

Animation captions:

1. timmy_age and age reference the same object.
2. Assigning to parameter age doesn't change timmy_age. Only the local namespace is updated.
3. To change timmy_age, the function should use a return value.

@zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

The semantics of passing object references as arguments is important because modifying an argument that is referenced elsewhere in the program may cause side effects outside of the function scope. When a function modifies a parameter, whether or not that modification is seen outside the scope of the function depends on the *mutability* of the argument object.

- If the object is **immutable**, such as a string or integer, then the modification is limited to inside the function. Any modification to an immutable object results in the creation of a *new* object in the function's local scope, thus leaving the original argument object unchanged.
- If the object is **mutable**, then in-place modification of the object can be seen outside the scope of the function. Any operation like adding elements to a container or sorting a list that is performed within a function will also affect any other variables in the program that reference the same object.

The following program illustrates how the modification of a list argument's elements inside a function persists outside of the function call.

**PARTICIPATION
ACTIVITY**

6.12.2: Modification of a list inside a function.

```

1 def modify(my_list):
2     my_list[1] = 99
3
4 my_list = [10, 20, 30]
5 modify(my_list)
6 print(my_list) # my_list still contains 99!
```

<< First < Back Step 1 of 7 Forward > Last >>

→ line that has just executed

→ next line to execute

Frames

Objects

Sometimes a programmer needs to pass a mutable object to a function but wants to make sure that the function does not modify the object at all. One method to avoid unwanted changes is to pass a copy of the object as the argument instead, like in the statement `my_func(my_list[:])`.

**PARTICIPATION
ACTIVITY**

6.12.3: Modification of a list inside a function.

```

1 def modify(my_list):
2     my_list[1] = 99 # Modifying only the copy
3
4 my_list = [10, 20, 30]
5 modify(my_list[:]) # Pass a copy of the list
6
7 print(my_list) # my_list does not contain 99!
```

@zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

→ line that has just executed

→ next line to execute

Frames

Objects

zyDE 6.12.1: List argument modification.

©zyBooks 03/05/20 10:28 591419

Alexey Munishkin

UCSCCSE20NawabWinter2020

Address the FIXME comments. Move the respective code from the while-loop to the create function. The add_grade function has already been created.

Note: split() and strip() are string methods further explained elsewhere. split() separates a string into tokens using any whitespace as the default separator. The tokens are returned as a list (i.e., 'a b c'.split() returns ['a', 'b', 'c']). strip() returns a copy of a string with leading and trailing whitespace removed.

[Load default template](#)

```

1 def add_grade(student_grades):
2     print('Entering grade. \n')
3     name, grade = input(grade_prompt).split()
4     student_grades[name] = grade
5
6 # FIXME: Create delete_name function
7
8 # FIXME: Create print_grades function
9
10 student_grades = {} # Create an empty dict
11 grade_prompt = "Enter name and grade (Ex. 'Bob A+'): \n"
12 delete_prompt = "Enter name to delete: \n"
13 menu_prompt = ("1. Add/modify student grade \n"
14               "2. Delete student grade \n"
15               "3. Print student grades \n"
16               "4. Quit \n \n")
17
18 command = input(menu_prompt).lower().strip()
19
20 while command != '4': # Exit when user enters '4'
21     if command == '1':

```

```

1
Johnny B+
1

```

Run

<

>

PARTICIPATION
ACTIVITY

6.12.4: Arguments and mutability.



- 1) Assignments to a parameter name inside a function affect the code outside the function.

☐ True
☐ False

- 2) When a function is called, copies of all the argument objects are made.

☐ True
☐ False

- 3) Adding an element to a dictionary argument in a function might affect



©zyBooks 03/05/20 10:28 591419

Alexey Munishkin

UCSCCSE20NawabWinter2020

variables outside the function that reference the same dictionary object.

- ☐ True
- ☐ False

4) A programmer can protect mutable arguments from unwanted changes by passing a copy of the object to a function.

- ☐ True
- ☐ False

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

CHALLENGE ACTIVITY

6.12.1: Change order of elements in function list argument.

Write a function `swap` that swaps the first and last elements of a list argument.

Sample output with input: 'all,good,things,must,end,here'

`['here', 'good', 'things', 'must', 'end', 'all']`

```
1
2 ''' Your solution goes here '''
3
4 values_list = input().split(',') # Program receives comma-separated values like 5,4,12,19
5 swap(values_list)
6
7 print(values_list)
```

Run

View solution ▼ (Instructors only)

↓ Download student submissions ⓘ

6.13 Keyword arguments and default parameter values

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Keyword arguments

Sometimes a function requires many arguments. In such cases, a function call can become very long and difficult to read. Furthermore, a programmer might easily make a mistake when calling such a function if the ordering of the arguments is given incorrectly. Consider the following program:

Figure 6.13.1: A function with many arguments.

```
def print_book_description(title, author, publisher, year, version, num_chapters, num_pages):
    # Format and print description of a book...

print_book_description('The Lord of the Rings', 'J. R. R. Tolkien', 'George Allen & Unwin',
                      1954, 1.0, 22, 456)
```

In the example above, a programmer might very easily swap the positions of some of the arguments in the function call, potentially introducing a bug into the program. Python provides for **keyword arguments** that allow arguments to map to parameters by name, instead of implicitly by position in the argument list. When using keyword arguments, the argument list does not need to follow a specific ordering.

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Figure 6.13.2: Using keyword arguments.

```
def print_book_description(title, author, publisher, year, version, num_chapters, num_pages):
    # Format and print description of a book...

print_book_description(title='The Lord of the Rings', publisher='George Allen & Unwin',
                      year=1954, author='J. R. R. Tolkien', version=1.0,
                      num_pages=456, num_chapters=22)
```

Keyword arguments provide a bit of clarity to potentially confusing function calls. Good practice is to use keyword arguments for any function containing more than approximately 4 arguments.

Keyword arguments can be mixed with positional arguments, provided that the keyword arguments come last. A common error is to place keyword arguments before all position arguments, which generates an exception.

Figure 6.13.3: All keyword arguments must follow positional arguments.

```
def split_check(amount, num_people, tax_percentage, tip_percentage):
    # ...

split_check(125.00, tip_percentage=0.15, num_people=2, tax_percentage=0.095)
```

PARTICIPATION ACTIVITY

6.13.1: Keyword arguments.

Assume the function below is defined:

```
def split_check(amount, num_people, tax_percentage, tip_percentage):
    # ...
```

- 1) What value is passed as the `tax_percentage` argument in the function call `split_check(60.52, 5, .07, tip_percentage=0.18)`?
Answer ERROR if an error occurs.

Check [Show answer](#)

- 2) What value is passed as the `num_people` argument in the function call `split_check(tax_percentage=.07, 60.52, 2, tip_percentage=0.18)`?
Answer ERROR if an error occurs.

Check [Show answer](#)

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Default parameter values

Sometimes a function has parameters that are optional. A function can have a **default parameter value** for one or more parameters, meaning that a function call can optionally omit an argument, and the default parameter value will be substituted for the corresponding omitted argument.

The following function prints a date in a particular style, given parameters for day, month, and year. The fourth parameter indicates the desired style, with 0 meaning American style, and 1 meaning European style. For July 30, 2012, the American style is 7/30/2012 and the European style is 30/7/2012.

Figure 6.13.4: Parameter with a default value.

```
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(30, 7, 2012, 0)
print_date(30, 7, 2012, 1)
print_date(30, 7, 2012) # style argument not provided! Default value of 0 used.
```

```
7 / 30 / 2012
30 / 7 / 2012
7 / 30 / 2012
```

The fourth (and last) parameter is defined with a default value: `style=0`. If the function call does not provide a fourth argument, then `style` has value 0. A parameter's **default value** is the value used in the absence of an argument in the function call.

The same can be done for other parameters, as in: `def print_date(day=1, month=1, year=2000, style=0)`. If positional arguments are passed (i.e., not keyword-arguments), then only the last arguments can be omitted. The following are valid calls to this `print_date()` function:

Figure 6.13.5: Valid function calls with default parameter values.

```
print_date(30, 7, 2012, 0) # Defaults: none
print_date(30, 7, 2012)   # Defaults: style=0
print_date(30, 7)         # Defaults: year=2000, style=0
print_date(30)            # Defaults: month=1, year=2000, style=0
print_date()              # Defaults: day=1, month=1, year=2000, style=0
```

If a parameter does not have a default value, then failing to provide an argument (either keyword or positional) generates an error.

A **common error** is to provide a mutable object, like a list, as a default parameter. Such a definition can be problematic because the default argument object is created only once, at the time the function is defined (when the script is loaded), and not every time the function is called. Modification of the default parameter object will persist across function calls, which is likely not what a programmer intended. The below program demonstrates the problem with mutable default objects and illustrates a solution that creates a new empty list each time the function is called.

Figure 6.13.6: Mutable default objects remain changed over multiple function calls.



Default object modification

```
def append_to_list(value, my_list=[]):  
    my_list.append(value)  
    return my_list  
  
numbers = append_to_list(50) #  
# default list appended with 50  
print(numbers)  
numbers = append_to_list(100) #  
# default list appended with 100  
print(numbers)
```

```
[50]  
[50, 100]
```

Solution: Make new list

```
def append_to_list(value, my_list=None): #  
    Use default parameter value of None  
    if my_list == None: # Create a new list  
        if a list was not provided  
            my_list = []  
  
    my_list.append(value)  
    return my_list  
  
numbers = append_to_list(50) # default list  
# appended with 50  
print(numbers)  
numbers = append_to_list(100) # default list  
# appended with 100  
print(numbers)
```

```
[50]  
[100]
```

The left program shows a function `append_to_list()` that has an empty list as default value of `my_list`. A programmer might expect that each time the function is called without specifying `my_list`, a new empty list will be created and the result of the function will be `[value]`. However, the default object persists across function calls. The solution replaces the default list with `None`, checking for that value, and then creating a new empty list in the local scope if necessary.

PARTICIPATION ACTIVITY

6.13.2: Default parameter values.

The following function is defined:

```
def split_check(amount, num_people, tax_percentage=0.095, tip_percentage=0.15)  
    # ...
```

- 1) What will the parameter `tax_percentage` be assigned for the following call? Type ERROR if the call is invalid.
`split_check(65.50, 3)`

[Check](#) [Show answer](#)

- 2) What will the parameter `tax_percentage` be assigned for the following call? Type ERROR if the call is invalid.
`split_check(65.50, 3, 0.125)`

[Check](#) [Show answer](#)

- 3) What will the parameter `num_people` be assigned for the following call? Type ERROR if the call is invalid.
`split_check(12.50, tip_percentage=0.18)`

[Check](#) [Show answer](#)

- 4) What will the parameter `num_people` be assigned for the following call? Type

@zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

@zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

ERROR if the call is invalid.

```
split_check(tip_percentage=0.18,  
12.50, 4)
```

Check [Show answer](#)

Mixing keyword arguments and default parameter values

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin

Mixing keyword arguments and default parameter values allows a programmer to omit arbitrary arguments from a function call. Because keyword arguments use names instead of position to match arguments to parameters, any argument can be omitted as long as that argument has a default value.

Consider the `print_date` function from above. If every parameter has a default value, then the user can use keyword arguments to pass specific arguments anywhere in the argument list. Below are some sample function calls:

Figure 6.13.7: Mixing keyword arguments and default parameter values allows omitting arbitrary arguments.

```
def print_date(day=1, month=1, year=2000, style=0):  
    # ...  
  
print_date(day=30, year=2012) # Defaults:      month=1,      style=0  
print_date(style=1)          # Defaults: day=1, month=1, year=2000  
print_date(year=2012, month=4) # Defaults: day=1,      style=0
```

PARTICIPATION ACTIVITY

6.13.3: Mixing keyword and default arguments.

Assume the function below is defined:

```
def split_check(amount=10, num_people=2, tax_percentage=0.095, tip_percentage=0.18):  
    # ...
```

When entering answers, use the same number of significant digits as the default parameter values in the `split_check()` definition.

- 1) What will the parameter `tax_percentage` be assigned for the following call? Type ERROR if the call is invalid. `split_check(amount=49.50, num_people=3)`

Check [Show answer](#)

- 2) Write a statement that splits a \$50 check among 4 people. Use the default tax percentage and tip amount.

Check [Show answer](#)

- 3) Write a statement that splits a \$25 check among 3 people and leaves a 25% tip. Use the default tax rate.

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Check Show answer

CHALLENGE
ACTIVITY

6.13.1: Return number of pennies in total.



Write a function `number_of_pennies()` that returns the total number of pennies given a number of dollars and (optionally) a number of pennies.

Sample output with inputs: 5 6 4

506

400

```
1
2 ''' Your solution goes here '''
3
4 print(number_of_pennies(int(input()), int(input()))) # Both dollars and pennies
5 print(number_of_pennies(int(input())))               # Dollars only
```

Run

View solution ▼ (Instructors only)

↓ Download student submissions ⓘ



CHALLENGE
ACTIVITY

6.13.2: Default parameters: Calculate splitting a check between diners.



Write a `split_check` function that returns the amount that each diner must pay to cover the cost of the meal.

The function has 4 parameters:

- `bill`: The amount of the bill.
- `people`: The number of diners to split the bill between.
- `tax_percentage`: The extra tax percentage to add to the bill.
- `tip_percentage`: The extra tip percentage to add to the bill.

The tax or tip percentages are optional and may not be given when calling `split_check`. Use default parameter values of 0.15 (15%) for `tip_percentage`, and 0.09 (9%) for `tax_percentage`.

Sample output with inputs: 25 2

Cost per diner: 15.5

Sample output with inputs: 100 2 0.075 0.21

Cost per diner: 64.25

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

```

1 # FIXME: Write the split_check function. HINT: Calculate the amount of tip and tax,
2 # add to the bill total, then divide by the number of diners.
3
4 ''' Your solution goes here '''
5
6 bill = float(input())
7 people = int(input())
8
9 # Cost per diner at the default tax and tip percentages
10 print('Cost per diner:', split_check(bill, people))
11
12 bill = float(input())
13 people = int(input())
14 new_tax_percentage = float(input())
15 new_tip_percentage = float(input())
16
17 # Cost per diner at different tax and tip percentages
18 print('Cost per diner:', split_check(bill, people, new_tax_percentage, new_tip_percentage))

```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Run

View solution ▼ (Instructors only)

↓ Download student submissions ⓘ

6.14 Arbitrary argument lists

Arbitrary arguments

Sometimes a programmer doesn't know how many arguments a function requires. A function definition can include a ***args** parameter that collects optional positional parameters into an **arbitrary argument list** tuple.

Figure 6.14.1: Arbitrary numbers of position arguments using *args.

```

def sandwich(bread, meat, *args):
    print('{} on {}'.format(meat, bread), end=' ')
    if len(args) > 0:
        print('with', end=' ')
        for extra in args:
            print(extra, end=' ')
        print("")
    sandwich('sourdough', 'turkey', 'mayo')
    sandwich('wheat', 'ham', 'mustard', 'tomato', 'lettuce')

```

turkey on sourdough with mayo
ham on wheat with mustard tomato lettuce

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Adding a final function parameter of ****kwargs** creates a dictionary containing "extra" arguments not defined in the function definition; kwargs is short for **keyword arguments**. The keys of the dictionary are the parameter names specified in the function call.

Figure 6.14.2: Arbitrary numbers of keyword arguments using **kwargs.

--	--

```
def sandwich(bread, meat, **kwargs):
    print('{} on {}'.format(bread, meat))
    for category, extra in kwargs.items():
        print(' {}: {}'.format(category, extra))

sandwich('sourdough', 'turkey', sauce='mayo')
sandwich('wheat', 'ham', sauce1='mustard', veggie1='tomato',
        veggie2='lettuce')
```

```
sourdough on
turkey
    sauce: mayo

wheat on ham
    veggie2:
lettuce
    sauce1:
mustard
    veggie1:
tomato
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

The `*` and `**` characters in `*args` and `**kwargs` are the important symbols. Using "args" and "kwargs" is standard practice, but any valid identifier is acceptable (like perhaps using `*condiments` in the sandwich example).

One or both of `*args` or `**kwargs` can be used. They must come last (and in that order if both are used) in the parameter list, otherwise an error occurs.

Below is a practical example showing how to combine normal parameters and the `**kwargs` parameter. Operating systems like Windows or MacOS have a command line that can be used instead of clicking icons on a desktop. To start an application using the command line, a user types in the application name followed by some options (usually denoted with a double dash `--`), as in `notepad.exe` or `firefox.exe --new-window=http://google.com --private-toggle=True`. The example below uses a function call's arguments to generate a new command.

PARTICIPATION ACTIVITY

6.14.1: Arbitrary numbers of arguments using `*args` and `**kwargs`.

```
1 def gen_command(application, **kwargs):
2     command = application
3
4     for argument in kwargs:
5         command += ' --{}={}'.format(argument, kwargs[argument])
6     return command
7
8 print(gen_command('notepad.exe')) # No options
9 print(gen_command('Powerpoint.exe', file='pres.ppt', start=True, sl
```

<< First < Back Step 1 of 19 Forward > Last >>

→ line that has just executed

→ next line to execute

Frames

Objects

PARTICIPATION ACTIVITY

6.14.2: Arbitrary arguments.

- 1) Complete the first line of the function definition for `f()` requiring two arguments `arg1` and `arg2`, and an arbitrary argument list `*args`.

```
def f( ):
    # ...
```

Check

Show answer

- 2) Complete the function call so that the output of the program is

```
John is:
age: 10
gender: m
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

```
def stats(name, **info):
    print(name, 'is:')
    for key, value in
info.items():
        print('{}:
{}'.format(key, value))

stats(

```

Check Show answer

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

6.15 Multiple function outputs

Multiple function outputs

Occasionally a function should produce multiple output values. However, function return statements are limited to returning only one value. A workaround is to package the multiple outputs into a single container, commonly a tuple, and to then return that container.

Figure 6.15.1: Multiple outputs can be returned in a container.

```
student_scores = [75, 84, 66, 99, 51, 65]

def get_grade_stats(scores):
    # Calculate the arithmetic mean
    mean = sum(scores)/len(scores)

    # Calculate the standard deviation
    tmp = 0
    for score in scores:
        tmp += (score - mean)**2
    std_dev = (tmp/len(scores))**0.5

    # Package and return average, standard deviation in
    a tuple
    return mean, std_dev

# Unpack tuple
average, standard_deviation =
get_grade_stats(student_scores)

print('Average score:', average)
print('Standard deviation:', standard_deviation)
```

Average score: 73.33333333333333
Standard deviation:
15.260697523012796

The above example calculates the mean and standard deviation of a set of student test scores. The statement `return mean, std_dev` creates and returns a tuple container. Recall that a tuple doesn't require parentheses around the contents, as the comma indicates a tuple should be created. An equivalent statement would have been `return (mean, std_dev)`. The outputs could also have been returned in a list, as in `return [mean, std_dev]`.

The statement `average, standard_deviation = get_grade_stats(student_scores)` utilizes **unpacking** to perform multiple assignments at once, so that `average` and `standard_deviation` are assigned the first and second elements from the returned tuple.

PARTICIPATION ACTIVITY

6.15.1: Multiple function outputs.

- 1) The statement `return a, b, [c, d]` is valid.

- True
☐ False
- 2) A function may return multiple objects.
☐ True
☐ False

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

6.16 Help! Using docstrings to document functions

Docstrings

A large program can contain many functions with a wide variety of uses. A programmer should document each function, giving a high-level description of the purpose of the function, so that later readers of the code can more easily understand. A **docstring** is a string literal placed in the first line of a function body.

A docstring starts and ends with three consecutive quotation marks. Good practice is to keep the docstring of a simple function as a single line, including the quotes. Furthermore, there should be no blank lines before or after the docstring.

Multi-line docstrings can be used for more complicated functions to describe the function arguments. Multi-line docstrings should use consistent indentation for each line, separating the ending triple-quotes by a blank line.

Figure 6.16.1: A single and a multi-line docstring.

```
def num_seats(airliner_type):  
    """Determines number of seats on a plane"""  
    #Function body statements ...  
  
def ticket_price(origin, destination, coach=True, first_class=False):  
    """Calculates the price of a ticket between two airports.  
    Only one of coach or first_class must be True.  
  
    Arguments:  
    origin -- string representing code of origin airport  
    destination -- string representing code of destination airport  
  
    Optional keyword arguments:  
    coach -- Boolean. True if ticket cost priced for a coach class ticket (default True)  
    first_class -- Boolean. True if ticket cost priced for a first class ticket (default False)  
    """  
    #Function body statements ...
```

The help() function

The **help()** function can aid a programmer by providing them with all the documentation associated with an object. A statement such as `help(ticket_price)` would print out the docstring for the `ticket_price()` function, providing the programmer with information about how to call that function.

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

zyDE 6.16.1: Using the help() function.

Run the following program that prints out the response of `help(ticket_price)`. Add an additional parameter `"vegetarian=False"` to `ticket_price`, augment the docstring appropriately, and run the program again.

Load default template...

Run

<

The `help()` function actually works with most of the built-in Python names, since the language creators were nice enough to provide docstrings for many items. Notice that the output of `help` depends on the object passed as an argument. If the argument is a function, then the docstring is printed. If you have studied classes or modules, note how `help(str)` prints out a description of the string str class methods and how `help(math)` prints out all the contents of the math module.

zyDE 6.16.2: Use the `help()` function on built-in names.

Use the following interpreter to play with the `help` function(). Try the following: `help(str)`, `help(range)`, and `help(max)`. Try defining a function or two of your own. The statement `help(__name__)` runs the `help` function on the global scope of the editor, printing information about all items defined there.

```
17 help(ticket_price)
```

Load default template...

Run

```
1 def my_function(arg):
2     pass
3
4 help(__name__)
5
```

6.17 Engineering examples

This section contains some examples of functions used to carry out engineering-type calculations.

Example: Gas equation

An equation used in physics and chemistry that relates pressure, volume, and temperature of a gas is $PV = nRT$. P is the pressure, V the volume, T the temperature, n the number of moles, and R a constant. The function below outputs the temperature of a gas given the other values.

Figure 6.17.1: $PV = nRT$. Compute the temperature of a gas.

```
gas_constant = 8.3144621 # Joules / (mol*Kelvin)

def convert_to_temp(pressure, volume, mols):
    """Convert pressure, volume, and moles to a
    temperature"""
    return (pressure * volume) / (mols * gas_constant)

press = float(input('Enter pressure (in Pascals): '))
vol = float(input('Enter volume (in cubic meters): '))
mols = float(input('Enter number of moles: '))

print('Temperature = {:.2f} K'.format(convert_to_temp(press,
vol, mols)))
```

```
Enter pressure (in Pascals):
2500
Enter volume (in cubic
meters): 35.5
Enter number of moles: 18
Temperature = 593.01 K
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

**PARTICIPATION
ACTIVITY**

6.17.1: $PV = nRT$ calculation.

Questions refer to convert_to_temp function above.

- 1) Function convert_to_temp uses a rewritten form of $PV = nRT$ to solve for T , namely $T = PV/nR$.
☐ True
☐ False
- 2) Function convert_to_temp uses a global variable for the gas constant R .
☐ True
☐ False
- 3) Function convert_to_pres() would likely return $(temp * volume) / (mols * gas_constant)$.
☐ True
☐ False

Example: Trajectory of object on Earth

Common physics equations determine the x and y coordinates of a projectile object at any time, given the object's initial velocity and angle at time 0 with initial position $x = 0$ and $y = 0$. The equation for x is $v * t * \cos(a)$. The equation for y is $v * t * \sin(a) - 0.5 * g * t * t$.

The program's code asks the user for the object's initial velocity, angle, and height (y position), and then prints the object's position for every second until the object's y position is no longer greater than 0 (meaning the object fell back to Earth).

zyDE 6.17.1: Trajectory of object on Earth.

Load default template...

45
100
3

Run

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

**PARTICIPATION
ACTIVITY**

6.17.2: Projective location.

Questions refer to function trajectory above.

- 1) trajectory() cannot return two values
(for x and y), so instead returns a single
tuple containing both x and y.

- ☐ True
☐ False

- 2) The program could replace float() by
int() without causing much change in
computed values.

- ☐ True
☐ False

- 3) Each iteration of the loop will see x_loc
increase.

- ☐ True
☐ False

- 4) Assuming the launch angle is less than
90 degrees, each iteration of the loop
will see x_loc increase.

- ☐ True
☐ False

```
1
2
3 import math
4
5 def trajectory(t, a, v, g, h):
6     """Calculates new x,y position"""
7     x = v * t * math.cos(a)
8     y = h + v * t * math.sin(a) - 0.5 * g * t**2
9     return (x,y)
10
11 def degrees_to_radians(degrees):
12     """Converts degrees to radians"""
13     return ((degrees * math.pi) / 180.0)
14
15 gravity = 9.81 # Earth gravity (m/s^2)
16 time = 1.0 # time (s)
17 x_loc = 0
18 h = 0
19
20 < >
```

**CHALLENGE
ACTIVITY**

6.17.1: Function to compute gas volume.

Define a function compute_gas_volume that returns the volume of a gas given parameters pressure, temperature, and moles. Use the gas equation $PV = nRT$, where P is pressure in Pascals, V is volume in cubic meters, n is number of moles, R is the gas constant 8.3144621 J / (mol*K), and T is temperature in Kelvin.

Sample output with inputs: 100.0 1.0 273.0

Gas volume: 22.698481533 m^3

Run

View solution ▼ (Instructors only)

↓ Download student submissions ⓘ

6.18 LAB: Miles to track laps

One lap around a standard high-school running track is exactly 0.25 miles. Write the function `miles_to_laps()` that takes a number of miles as an argument and returns the number of laps. Complete the program to output the number of laps.

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:
`print('{:.2f}'.format(your_value))`

Ex: If the input is:

1.5

the output is:

6.00

Ex: If the input is:

2.2

the output is:

8.80

Your program must define and call the following function:

```
def miles_to_laps(user_miles)
```

LAB
ACTIVITY

6.18.1: LAB: Miles to track laps

0 / 10

main.py

Load default template...

```

1 ''' Define your function here '''
2
3 if __name__ == '__main__':
4     ''' Type your code here. Your code must call the function. '''

```

Develop mode
Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)

→

main.py
(Your program)

→

Output (shown below)

Program output displayed here

Lab statistics and submissions
Show

Solution
Show

Tests
Show

6.19 LAB: Max magnitude

Write a function `max_magnitude()` with two integer input parameters that returns the largest magnitude value. Use the function in a program that takes two integer inputs, and outputs the largest magnitude value.

Ex: If the inputs are:

5 7

the function returns:

7

Ex: If the inputs are:

-8 -2

the function returns:

-8

Note: The function does not just return the largest value, which for -8 -2 would be -2. Though not necessary, you may use the built-in absolute value function to determine the max magnitude, but you must still output the input number (Ex: Output -8, not 8).

Your program must define and call the following function:

```
def max_magnitude(user_val1, user_val2)
```

LAB
ACTIVITY

6.19.1: LAB: Max magnitude

0 / 10

03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

main.py

Load default template...

```
1 import math
2
3 ''' Define your function here. '''
4
5 if __name__ == '__main__':
6     ''' Type your code here. '''
7 |
```

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)

→

main.py
(Your program)

→

Output (shown below)

Program output displayed here

Lab statistics and submissions

Show ▾

Solution

Show ▾

Tests

Show ▾



6.20 LAB: Driving costs - functions

Driving is expensive. Write a program with a car's miles/gallon and gas dollars/gallon (both floats) as input, and output the gas cost for 10 miles, 50 miles, and 400 miles.

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:

```
print('{:.2f}'.format(your_value))
```

Ex: If the input is:

```
20.0
3.1599
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

the output is:

```
1.58
7.90
63.20
```

Your program must define and call the following `driving_cost()` function. Given input parameters `driven_miles`, `miles_per_gallon`, and `dollars_per_gallon`, the function returns the dollar cost to drive those miles.

Ex: If the function is called with:

```
50 20.0 3.1599
```

the function returns:

```
7.89975
```

```
def driving_cost(driven_miles, miles_per_gallon, dollars_per_gallon)
```

Your program should call the function three times to determine the gas cost for 10 miles, 50 miles, and 400 miles.

Note: This is a lab from a previous chapter that now requires the use of a function.

LAB
ACTIVITY

6.20.1: LAB: Driving costs - functions

0 / 10

main.py

Load default template...

```
1 ''' Define your function here. '''
2
3 if __name__ == '__main__':
4     ''' Type your code here. '''
5
6
7 |
```

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

If your code requires input values, provide them here.

Run program

Input (from above) →

main.py
(Your program)

→ Output (shown below)

Program output displayed here

Lab statistics and submissions

Show ▼

Solution

Show ▼

Tests

Show ▼

©zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020

6.21 LAB: Step counter

A pedometer treats walking 2,000 steps as walking 1 mile. Write a program whose input is the number of steps, and whose output is the miles walked.

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:
`print('{:.2f}'.format(your_value))`

Ex: If the input is:

5345

the output is:

2.67

Your program must define and call the following function. The function should return the amount of miles walked.
`def steps_to_miles(user_steps)`

LAB
ACTIVITY

6.21.1: LAB: Step counter

0 / 10

main.py

Load default template...

©zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020

```

1 ''' Define your function here '''
2
3 if __name__ == '__main__':
4     ''' Type your code here. '''

```

Develop mode
Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program
Input (from above) →
main.py
(Your program)
→
Output (shown below)

Program output displayed here

Lab statistics and submissions
Show ▼

Solution
Show ▼

Tests
Show ▼

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

6.22 LAB: A jiffy

A "jiffy" is the scientific name for 1/100th of a second. Given an input number of seconds, output the number of "jiffies."

Output each floating-point value with two digits after the decimal point, which can be achieved as follows:

```
print('{:.2f}'.format(your_value))
```

Ex: If the input is:

15

the output is:

1500.00

Your program must define and call the following function. The function should return the number of "jiffies".

```
def seconds_to_jiffies(user_seconds)
```

LAB
ACTIVITY
6.22.1: LAB: A jiffy
0 / 10

main.py
Load default template...

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

```

1  ''' Define your function here'''
2
3  if __name__ == '__main__':
4      ''' Type your code here. Your code must call the function. '''
5

```

©zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)



main.py
(Your program)



Output (shown below)

Program output displayed here

Lab statistics and submissions

Show ▾

Solution

Show ▾

Tests

Show ▾

6.23 LAB: Leap year - functions

A year in the modern Gregorian Calendar consists of 365 days. In reality, the earth takes longer to rotate around the sun. To account for the difference in time, every 4 years, a leap year takes place. A leap year is when a year has 366 days: An extra day, February 29th. The requirements for a given year to be a leap year are:

- 1) The year must be divisible by 4
- 2) If the year is a century year (1700, 1800, etc.), the year must be evenly divisible by 400

Some example leap years are 1600, 1712, and 2016.

Write a program that takes in a year and determines whether that year is a leap year.

Ex: If the input is:

1712

the output is:

©zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020

1712 is a leap year.

Ex: If the input is:

1913

the output is:

1913 is not a leap year.

Your program must define and call the following function. The function should return true if the input year is a leap year and false otherwise.

```
def is_leap_year(user_year)
```

Note: This is a lab from a previous chapter that now requires the use of a function.

LAB
ACTIVITY

6.23.1: LAB: Leap year - functions

0 / 10

main.py

Load default template...

```
1 ''' Define your function here. '''
2
3 if __name__ == '__main__':
4     ''' Type your code here. Your code must call the function. '''
```

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) →

main.py
(Your program) →

Output (shown below)

Program output displayed here

Lab statistics and submissions

Show ▾

Solution

Show ▾

Tests

Show ▾

6.24 LAB: Convert to binary - functions

Write a program that takes in a positive integer as input, and outputs a string of 1's and 0's representing the integer in binary. For an integer x , the algorithm is:

```
As long as x is greater than 0
    Output x % 2 (remainder is either 0 or 1)
    x = x // 2
```

Note: The above algorithm outputs the 0's and 1's in reverse order. You will need to write a second function to reverse the string.

Ex: If the input is:

6

the output is:

110

Your program must define and call the following two functions. The function `integer_to_reverse_binary()` should return a string of 1's and 0's representing the integer in binary (in reverse). The function `reverse_string()` should return a string representing the input string in reverse.

```
def integer_to_reverse_binary(integer_value)
def reverse_string(input_string)
```

Note: This is a lab from a previous chapter that now requires the use of a function.

LAB
ACTIVITY

6.24.1: LAB: Convert to binary - functions

0 / 10

main.py

Load default template...

```
1 ''' Define your function here. '''
2
3 if __name__ == '__main__':
4     ''' Type your code here. Your code must call the function. '''
5
6 |
```

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Input (from above) → **main.py** (Your program) → Output (shown below)

Program output displayed here

Lab statistics and submissions

Solution

Tests

[Show](#) ▼

[Show](#) ▼

[Show](#) ▼

6.25 LAB: Swapping variables

Write a program whose input is two integers and whose output is the two integers swapped.

Ex: If the input is:

3
8

the output is:

8 3

Your program must define and call the following function. `swap_values()` returns the two values in swapped order.

```
def swap_values(user_val1, user_val2)
```

LAB ACTIVITY 6.25.1: LAB: Swapping variables

0 / 10

main.py

[Load default template...](#)

```

1 ''' Define your function here. '''
2
3 if __name__ == '__main__':
4     ''' Type your code here. Your code must call the function. '''

```

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input

values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → Output (shown below)

Program output displayed here

Lab statistics and submissions [Show](#) ▾

Solution [Show](#) ▾

Tests [Show](#) ▾

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

6.26 LAB: Exact change - functions

Write a program with total change amount as an integer input that outputs the change using the fewest coins, one coin type per line. The coin types are dollars, quarters, dimes, nickels, and pennies. Use singular and plural coin names as appropriate, like 1 penny vs. 2 pennies.

Ex: If the input is:

0

or less, the output is:

no change

Ex: If the input is:

45

the output is:

1 quarter
2 dimes

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Your program must define and call the following function. The function `exact_change()` should return `num_dollars`, `num_quarters`, `num_dimes`, `num_nickels`, and `num_pennies`.

```
def exact_change(user_total)
```

Note: This is a lab from a previous chapter that now requires the use of a function.



main.py
Load default template...

```

1  ''' Define your function here '''
2
3  if __name__ == '__main__':
4      input_val = int(input())
5      num_dollars, num_quarters, num_dimes, num_nickels, num_pennies = exact_change(input_val)
6
7      ''' Type your code here. Your code must call the function. '''
8

```

©zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

→

main.py
 (Your program)

→

Output (shown below)

Program output displayed here

Lab statistics and submissions
Show ▼

Solution
Show ▼

Tests
Show ▼

6.27 LAB: Even/odd values in a list

©zyBooks 03/05/20 10:28 591419
 Alexey Munishkin
 UCSCCSE20NawabWinter2020

Write a program that reads a list of integers, and outputs whether the list contains all even numbers, odd numbers, or neither. The input begins with an integer indicating the number of integers that follow.

Ex: If the input is:

5
 2
 4
 6

```
8
10
```

the output is:

```
all even
```

Ex: If the input is:

```
5
1
3
5
7
9
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

the output is:

```
all odd
```

Ex: If the input is:

```
5
1
2
3
4
5
```

the output is:

```
not even or odd
```

Your program must define and call the following two functions. `is_list_even()` returns true if all integers in the list are even and false otherwise. `is_list_odd()` returns true if all integers in the list are odd and false otherwise.

```
def is_list_even(my_list)
def is_list_odd(my_list)
```

LAB
ACTIVITY

6.27.1: LAB: Even/odd values in a list

0 / 10



main.py

[Load default template...](#)

```
1 ''' Define your function here '''
2
3 if __name__ == '__main__':
4     ''' Type your code here. '''
5 |
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)

→

main.py
(Your program)

→

Output (shown below)

Program output displayed here

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Lab statistics and submissions

Show ▾

Solution

Show ▾

Tests

Show ▾

6.28 LAB: Output values in a list below a user defined amount - functions

Write a program that first gets a list of integers from input. The input begins with an integer indicating the number of integers that follow. Then, get the last value from the input, and output all integers less than or equal to that value.

Ex: If the input is:

```
5
50
60
140
200
75
100
```

the output is:

```
50
60
75
```

The 5 indicates that there are five integers in the list, namely 50, 60, 140, 200, and 75. The 100 indicates that the program should output all integers less than or equal to 100, so the program outputs 50, 60, and 75.

Such functionality is common on sites like Amazon, where a user can filter results.

Your code must define and call the following two functions:

```
def get_user_values()
def output_ints_less_than_or_equal_to_threshold(user_values, upper_threshold)
```

Utilizing functions will help to make your main very clean and intuitive.

Note: This is a lab from a previous chapter that now requires the use of functions.



main.py

[Load default template...](#)

```
1 ''' Define your function here '''
2
3 if __name__ == '__main__':
4     user_values, upper_threshold = get_user_values()
5     output_ints_less_than_or_equal_to_threshold(user_values, upper_threshold)
6 |
```

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Develop mode**Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)

**main.py**
(Your program)

Output (shown below)

Program output displayed here

Lab statistics and submissions

[Show](#) ▼

Solution

[Show](#) ▼

Tests

[Show](#) ▼

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

6.29 LAB: Warm up: Text analyzer & modifier

(1) Prompt the user to enter a string of their choosing. Output the string. (1 pt)

Ex:

```
Enter a sentence or phrase:
The only thing we have to fear is fear itself.
```

You entered: The only thing we have to fear is fear itself.

(2) Complete the `get_num_of_characters()` function, which returns the number of characters in the user's string. We encourage you to use a `for` loop in this function. (2 pts)

(3) Extend the program by calling the `get_num_of_characters()` function and then output the returned result. (1 pt)

(4) Extend the program further by implementing the `output_without_whitespace()` function.

`output_without_whitespace()` outputs the string's characters except for whitespace (spaces, tabs). Note: A tab is `'\t'`.

Call the `output_without_whitespace()` function in `main()`. (2 pts)

©zyBooks 03/05/20 10:28 591419

Alexey Munishkin

UCSCCSE20NawabWinter2020

Ex:

Enter a sentence or phrase:

The only thing we have to fear is fear itself.

You entered: The only thing we have to fear is fear itself.

Number of characters: 46

String with no whitespace: Theonlythingwehavetofearisfearitself.

LAB
ACTIVITY

6.29.1: LAB: Warm up: Text analyzer & modifier

0 / 6

main.py

[Load default template...](#)

```
1 def get_num_of_characters(input_str):
2     # Type your code here
3     return
4
5
6 if __name__ == '__main__':
7     # Type your code here
```

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)



main.py
(Your program)



Output (shown below)

Program output displayed here

Lab statistics and submissions

Show ▾

©zyBooks 03/05/20 10:28 591419

Alexey Munishkin

UCSCCSE20NawabWinter2020

Solution

Show ▾

Tests

Show ▾

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

6.30 LAB*: Program: Authoring assistant

(1) Prompt the user to enter a string of their choosing. Store the text in a string. Output the string. (1 pt)

Ex:

```
Enter a sample text:
we'll continue our quest in space.  there will be more shuttle flights and more
shuttle crews and,  yes;  more volunteers, more civilians,  more teachers in space.
nothing ends here;  our hopes and our journeys continue!

You entered: we'll continue our quest in space.  there will be more shuttle flights
and more shuttle crews and,  yes;  more volunteers, more civilians,  more teachers
in space.  nothing ends here;  our hopes and our journeys continue!
```

(2) Implement a `print_menu()` function, which has a string as a parameter, outputs a menu of user options for analyzing/editing the string, and returns the user's entered menu option and the sample text string (which can be edited inside the `print_menu()` function). Each option is represented by a single character.

If an invalid character is entered, continue to prompt for a valid choice. *Hint: Implement the Quit menu option before implementing other options.* Call `print_menu()` in the main section of your code. Continue to call `print_menu()` until the user enters q to Quit. (3 pts)

Ex:

```
MENU
c - Number of non-whitespace characters
w - Number of words
f - Fix capitalization
r - Replace punctuation
s - Shorten spaces
q - Quit

Choose an option:
```

(3) Implement the `get_num_of_non_WS_characters()` function. `get_num_of_non_WS_characters()` has a string parameter and returns the number of characters in the string, excluding all whitespace. Call `get_num_of_non_WS_characters()` in the `print_menu()` function. (4 pts)

©zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Ex:

```
Number of non-whitespace characters: 181
```

(4) Implement the `get_num_of_words()` function. `get_num_of_words()` has a string parameter and returns the number of words in the string. *Hint: Words end when a space is reached except for the last word in a sentence.* Call `get_num_of_words()` in the `print_menu()` function. (3 pts)

Ex:

```
Number of words: 35
```

(5) Implement the `fix_capitalization()` function. `fix_capitalization()` has a string parameter and returns an updated string, where lowercase letters at the beginning of sentences are replaced with uppercase letters. `fix_capitalization()` also returns the number of letters that have been capitalized. Call `fix_capitalization()` in the `print_menu()` function, and then output the number of letters capitalized and the edited string. *Hint 1: Look up and use Python functions `islower()` and `upper()` to complete this task. Hint 2: Create an empty string and use string concatenation to make edits to the string.* (3 pts)

Ex:

```
Number of letters capitalized: 3
Edited text: We'll continue our quest in space.  There will be more shuttle flights
and more shuttle crews and,  yes;  more volunteers, more civilians,  more teachers
in space.  Nothing ends here;  our hopes and our journeys continue!
```

(6) Implement the `replace_punctuation()` function. `replace_punctuation()` has a string parameter and two keyword argument parameters **`exclamation_count`** and **`semicolon_count`**. `replace_punctuation()` updates the string by replacing each exclamation point (!) character with a period (.) and each semicolon (;) character with a comma (,). `replace_punctuation()` also counts the number of times each character is replaced and outputs those counts. Lastly, `replace_punctuation()` returns the updated string. Call `replace_exclamation()` in the `print_menu()` function, and then output the edited string. (3 pts)

Ex:

```
Punctuation replaced
exclamation\_count: 1
semicolon\_count: 2
Edited text: we'll continue our quest in space.  there will be more shuttle flights
and more shuttle crews and,  yes,  more volunteers, more civilians,  more teachers
in space.  nothing ends here,  our hopes and our journeys continue.
```

(7) Implement the `shorten_space()` function. `shorten_space()` has a string parameter and updates the string by replacing all sequences of 2 or more spaces with a single space. `shorten_space()` returns the string. Call `shorten_space()` in the `print_menu()` function, and then output the edited string. *Hint: Look up and use Python function `isspace()`.* (3 pt)

Ex:

```
Edited text: we'll continue our quest in space. there will be more shuttle flights
and more shuttle crews and, yes, more volunteers, more civilians, more teachers in
space. nothing ends here; our hopes and our journeys continue!
```

LAB
ACTIVITY

6.30.1: LAB*: Program: Authoring assistant

0 / 20

main.py

[Load default template...](#)

03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

```

1 # Type all other functions here
2
3 def print_menu(usr_str):
4     menu_op = ''
5     # Complete print_menu() function
6     return menu_op, usr_str
7
8
9 if __name__ == '__main__':
10     # Complete main section of code

```

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

@zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)



main.py
(Your program)



Output (shown below)

Program output displayed here

Lab statistics and submissions

Show ▾

Solution

Show ▾

Tests

Show ▾



@zyBooks 03/05/20 10:28 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020