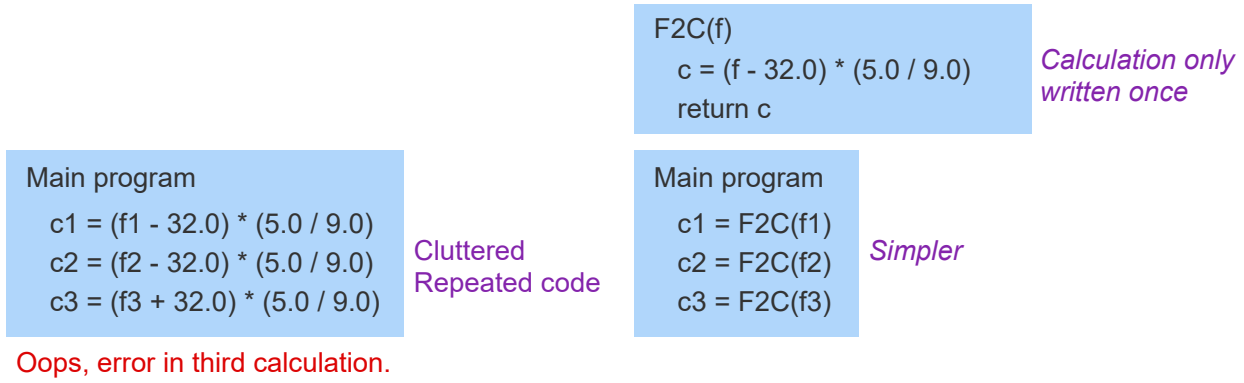
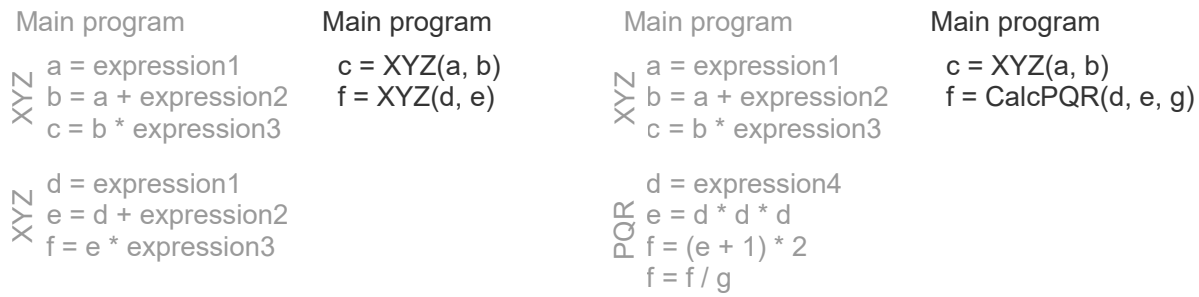
6.1 User-defined function basics

Functions (general)

A program may perform the same operation repeatedly, causing a large and confusing program due to redundancy. Program redundancy can be reduced by creating a grouping of predefined statements for repeated operations, known as a ***function***. Even without redundancy, functions can prevent a main program from becoming large and confusing.

PARTICIPATION ACTIVITY 6.1.1: Functions can reduce redundancy and keep the main program simple.





Animation content:

Static figure: Three code blocks are displayed, a main program, a function program, and a simplified version of the main program.
Main program
Begin Python code:

```
c1 = (f1 - 32.0) * (5.0 / 9.0)
c2 = (f2 - 32.0) * (5.0 / 9.0)
```

```
    c3 = (f3 + 32.0) * (5.0 / 9.0)
End Python code.

Begin Python code:
    c = (f - 32.0) * (5.0 / 9.0)
    return c
End Python code.
```



```
Main program
Begin Python code:
    c1 = F2C(f1)
    c2 = F2C(f2)
    c3 = F2C(f3)
End Python code.
```

Step 1: Commonly, a program performs the same operation, such as a calculation, in multiple places. Here, the Fahrenheit to Celsius calculation is done in three places.

```
Main program
    c1 = (f1 - 32.0) * (5.0 / 9.0)
    c2 = (f2 - 32.0) * (5.0 / 9.0)
    c3 = (f3 + 32.0) * (5.0 / 9.0)
```

Step 2: Repeated operations clutter the main program and are more prone to errors. The main program code is cluttered and has repeated code. In addition, the calculation for c3 has an error that uses the wrong operation.

Step 3: A better approach defines the Fahrenheit to Celsius calculation once, named F2C. Then, F2C can be "called" three times, yielding a simpler main program. A function is defined where the calculation is only written once.

```
F2C(f)
    c = (f - 32.0) * (5.0 / 9.0)
    return c
```

The simpler version of the main program code becomes

```
    c1 = F2C(f1)
    c2 = F2C(f2)
    c3 = F2C(f3)
```


Step 4: The impact is greater when the operation has multiple statements - here three statements but often tens of statements. The main program is much simpler. Two blocks appear, the first block titled Main program contains the code:

```
a = expression1
b = a + expression2
c = b * expression3
```

d = expression1
e = d + expression2
f = e * expression3
The label XYZ appears next to variables a, b, and c, and again next to variables d, e, and f.
The second block titled Main program contains the simpler code using a function:
c = XYZ(a, b)
f = XYZ(d, e)

Step 5: Even without repeated operations, calling predefined operations keeps the main program simple and intuitive. Two blocks appear, the first block titled Main program contains the code:
a = expression1
b = a + expression2
c = b * expression3

d = expression4
e = d * d * d
f = (e + 1) * 2
f = f / g
The label XYZ appears next to variables a, b, and c. The label PQR appears next to variables d, e, and f.
The second block titled Main program contains the simpler code:
c = XYZ(a, b)
f = CalcPQR(d, e, g)

Animation captions:

1. Commonly, a program performs the same operation, such as a calculation, in multiple places. Here, the Fahrenheit to Celsius calculation is done in three places.
2. Repeated operations clutter the main program and are more prone to errors.
3. A better approach defines the Fahrenheit to Celsius calculation once, named F2C. Then, F2C can be "called" three times, yielding a simpler main program.
4. The impact is greater when the operation has multiple statements - here three statements but often tens of statements. The main program is much simpler.
5. Even without repeated operations, calling predefined operations keeps the main program simple and intuitive.

PARTICIPATION ACTIVITY 6.1.2: Reasons for functions.

Consider the animation above.

1) In the original main program, the Fahrenheit to Celsius calculation

appeared how many times?

- 1
- 3

2) Along with yielding a simpler main program, using the predefined Fahrenheit to Celsius calculation prevented what error in the original program?

- Adding rather than subtracting 32.0
- Multiplying by 9.0 / 5.0 rather than by 5.0 / 9.0

3) In the last example above, the main program was simplified by ____.

- eliminating redundant code for operation XYZ
- predefining operations for XYZ and CalcPQR

Function basics

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

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

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 calc_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 get_name or calc_area.

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

PARTICIPATION ACTIVITY 6.1.3: Function example: Printing a pizza area.

```
def calc_pizza_area():
 pi_val = 3.14159265

 pizza_diameter = 12.0
 pizza_radius = pizza_diameter / 2.0
 pizza_area = pi_val * pizza_radius * pizza_radius
 return pizza_area

print(f'{12:.1f} inch pizza is {calc_pizza_area():.3f}
  square inches')
```

```
12.0 inch pizza is 113.097 square inches
```


Animation content:

Static figure:
Begin Python code:
def calc_pizza_area():
  pi_val = 3.14159265

  pizza_diameter = 12.0
  pizza_radius = pizza_diameter / 2.0
  pizza_area = pi_val * pizza_radius * pizza_radius
  return pizza_area

print(f'{12:.1f} inch pizza is {calc_pizza_area():.3f} square inches')
End Python code.

Step 1: The function call to calc_pizza_area() jumps execution to the function's statements. Each line within the calc_pizza_area function is highlighted one by one in order, up to return pizza_area.

Step 2: After the last statement of the calc_pizza_area() function, execution returns to the original location and the area of the pizza is returned and printed. A monitor displays the print statement, stating 12.0 inch pizza is 113.097 square inches.

Animation captions:

1. The function call to calc_pizza_area() jumps execution to the function's statements.
2. After the last statement of the calc_pizza_area() function, execution returns to the original location and the area of the pizza is returned and printed.

PARTICIPATION ACTIVITY 6.1.4: Function basics.

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

```
print(f'{12:.1f} inch pizza is {calc_pizza_area():.3f} square inches')
print(f'{12:.1f} inch pizza is {calc_pizza_area():.3f} square inches')
```

1) How many function calls to calc_pizza_area() exist?

Check Show answer

2) How many function definitions of calc_pizza_area() exist?

Check Show answer

3) How many output statements would execute in total?

Check Show answer

4) How many print statements exist in calc_pizza_area()?

Check Show answer

PythonTutor: Calling a function.

```
1 def get_face():
2     face_char = 'o   o\n  o\nooooo'
3     return face_char
4
5 print('Say cheese!')
6
7 print(get_face())
8
9 print('Did it turn out ok?')
```


line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 8

Program output

CHALLENGE ACTIVITY 6.1.1: Basic function call

get_pattern() returns 5 characters. Call get_pattern() twice in print() statements to return and print 10 characters. Example output:

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

Learn how our autograder works

566436.4601014.qx3zqy7


```
1 def get_pattern():
2     return '*****'
3
4 ''' Your solution goes here '''
5
```

Run

View your last submission

Return statements

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

PARTICIPATION ACTIVITY 6.1.5: Function example: Returning a value.

```
def compute_square(num_to_square):
    return num_to_square * num_to_square


num_squared = compute_square(7)

print(f'7 squared is {num_squared}')
```

7 squared is 49

Animation content:

Static figure:
Begin Python code:
def compute_square(num_to_square):
    return num_to_square * num_to_square

num_squared = compute_square(7)
print(f'7 squared is {num_squared}')
End Python code.

Step 1: Call compute_square and pass in the value 7. Compute_square has one parameter named num_to_square.

Step 2: Compute the square of num_to_square and return the result. The value passed to compute_square is 7, which is the value assigned to num_to_square. Compute_square returns num_to_square times num_to_square, so 7 times 7 is calculated, resulting in 49 as the return value.
Step 3: num_squared is assigned the return value of compute_square(7). A monitor displays the print statement, stating 7 squared is 49.

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 return only one item, not two or more (though a list or a tuple 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.

PARTICIPATION ACTIVITY 6.1.6: Return basics.

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

```
def compute_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 compute_cube(num):

```

Check Show answer

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 `calc_pizza_area(12.0)` or as `calc_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`.

PythonTutor: Function parameters.

```
1 import math
2 def calc_pizza_area(pizza_diameter):
3     pizza_radius = pizza_diameter / 2.0
4     pizza_area = math.pi * pizza_radius * pizza_radius
5     return pizza_area
6
7 print(f'12.0 inch pizza is {calc_pizza_area(12.0):.3f} square
8 print(f'16.0 inch pizza is {calc_pizza_area(16.0):.3f} square
```

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 14

Program output

Frames Objects

PARTICIPATION ACTIVITY 6.1.7: Parameters.

1) Complete the function definition to have a parameter named num_grade.

```
def get_letter_grade(
    ):
```

Check Show answer

2) Call a function named calc_calories() 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 get_birthday_age(user_age): simply returns the value of user_age + 1. What will the following code output?

```
print(get_birthday_age(42),
get_birthday_age(20))
```

Check Show answer

Multiple or no parameters

A function may have multiple parameters separated by commas. Parameters are assigned with argument values: First parameter with the first argument, second with the second, etc.

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

Figure 6.1.1: Function with multiple parameters.

```
def calc_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
   return pizza_volume

print(f'12.0 x 0.3 inch pizza is {calc_pizza_volume(12.0, 0.3):.3f} cubic
inches.')

print(f'12.0 x 0.8 inch pizza is {calc_pizza_volume(12.0, 0.8):.3f} cubic
inches.')

print(f'16.0 x 0.8 inch pizza is {calc_pizza_volume(16.0, 0.8):.3f} cubic
inches.')
```



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

PARTICIPATION ACTIVITY 6.1.8: Multiple parameters.

1) Which answer correctly defines two parameters `x` and `y` for a function definition:
`def calc_val(...):`?

   - (x; y)
   - (x y)
   - (x, y)

2) Which answer 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):`,
b is assigned with what value 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)

PARTICIPATION ACTIVITY 6.1.9: Calls with multiple parameters.

Given:

```
def compute_sum(num1, num2):
    return num1 + num2
```

1) What will be returned for the following function call?
`compute_sum(1, 2)`

Check Show answer

2) Write a function call using compute_sum() to return the sum of x and 400 (providing the arguments in that order).

Check Show answer

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, in which the input() function is called and evaluates to a value that is then passed as an argument to the int() function.

PythonTutor: 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 calc_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
12     return total_calories
13
14
15 print(f'12 inch pizza has {calc_pizza_calories(12.0):.2f} calories
16 print(f'14 inch pizza has {calc_pizza_calories(14.0):.2f} calories
```


line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 26

Program output

Frames Objects

PARTICIPATION ACTIVITY 6.1.10: Hierarchical function calls.

Complete the calc_pizza_calories_per_slice() function to calculate the calories for a single slice of pizza.

A calc_pizza_calories() function returns a pizza's total calories given the pizza diameter passed as an argument.

A calc_num_pizza_slices() function returns the number of slices in a pizza given the pizza diameter passed as an argument.

```
def calc_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 calculate the total calories for a pizza with diameter `pizza_diameter`.

`total_calories =`

Check Show answer

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

`calories_per_slice =`

Check Show answer

CHALLENGE ACTIVITY 6.1.2: Basic function call.

Complete the function definition to return the hours given minutes.

Sample output with input: 210.0

```
3.5
```

Learn how our autograder works

566436.4601014.qx3zqy7


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

Run

View your last submission

CHALLENGE ACTIVITY 6.1.3: Enter the output of the returned value.

566436.4601014.qx3zqy7

Start

Type the program's output

```
def change_value(x):
    return x + 1

print(change_value(4))
```

5

1 2

Check Next

CHALLENGE ACTIVITY 6.1.4: Functions with parameters and return values.

566436.4601014.qx3zqy7

Start

Define a function calculate_result() that has one parameter and returns the parameter minus 8.7.

Ex: If the input is `5.3`, then the output is:

```
-3.40
```

```
1
2 ''' Your code goes here '''
3
4 input_val = float(input())
5
6 print(f'{calculate_result(input_val):.2f}')
```

1 2

Check Next level

6.2 Print functions

Printing from a function

A common operation for a function is to print text. Large text outputs can clutter the main program, especially if the text needs to be output multiple times. A function that only prints typically does not return a value. A function with no return statement is called a ***void function***, and such a function returns the value `None`.

PARTICIPATION ACTIVITY 6.2.1: Printing with a void function.

```
def print_summary(oid, items, price):
    print(f'Order {oid}:')
    print(f'   Items: {items}')
    print(f'   Total: ${price:.2f}')



...
# Assume oid = 42, items = 4, price = 13.99
print_summary(oid, items, price)

# Continues execution
...
```

```
Order 42:
   Items: 4
   Total: $13.99
```

Animation content:

Static figure: A code block and a monitor are displayed.
Begin Python code:
Def print_summary(oid, items, price):
print(f'Order {oid}:')
print(f' Items: {items}')
print(f' Total: ${price:.2f}')

...
# Assume oid = 42, items = 4, price = 13.99
print_summary(oid, items, price)
...

Continues execution
End Python code.

Step 1: Printing instructions can clutter a program. Here, the program consists of three print statements.
Step 2: A print function can handle output and reduce clutter in the main program. The three print statements are moved from the main program to a new function called print_summary(oid, items, price). print_summary() takes three parameters, which are all used by the print statements. A function call print_summary(oid, items, price) replaces the print statements in the main program.
Step 3: The main program calls function print_summary(), which prints the parameters as formatted output. A code comment states to assume oid = 42, items = 4, and price = 13.99. A monitor displays the print statements, stating
Order 42:
Items: 4
Total: $13.99
Step 4: print_summary() completes execution and returns back to the caller, the main program.

Animation captions:

1. Printing instructions can clutter a program.
2. A print function can handle output and reduce clutter in the main program.
3. The main program calls function print_summary(), which prints the parameters as formatted output.
4. print_summary() completes execution and returns back to the caller, the main program.

PARTICIPATION ACTIVITY 6.2.2: Print functions.

1) Print operations must be performed in the main program.

- True
- False

2) A void function can have any number of parameters.

- True
- False

3) A print function must return the value that was output.

- True
- False

A function that produces output can also return a value, but this material separates these operations for clarity. A function that both outputs and returns a value is not void.

Calling a print function multiple times

One benefit of a print function is that complex output statements can be written in code once. Then the print function can be called multiple times to produce the output instead of rewriting complex statements for every necessary instance. Changes to output and formatting are made easier and are less prone to error.

PARTICIPATION ACTIVITY 6.2.3: Calling a print function repeatedly.


```
...
#·Read from input: word = 'Show'
print('--The Greatest *', end='')
print(word, end='')
print('* on Earth!--')

# Read from input: word = 'Song'
print('--The Greatest *', end='')
print(word, end='')
print('* on Earth!--')
```

```
def print_greatest(word):
    print('--The Greatest *', end='')
    print(word, end='')
    print('* on Earth!--')


...
# Read from input: word = 'Show'
print_greatest(word)
...
...
# Read from input: word = 'Song'
```

```
print_greatest(word)
```

Program A

Program B

Animation content:

Static figure: Two code blocks are displayed.
Program A:
Begin Python code:
…
# Read from input: word = 'Show'
print('--The Greatest *', end='')
print(word, end='')
print('* on Earth!--')

# Read from input: word = 'Song'
print('--The Greatest *', end='')
print(word, end='')
print('* on Earth!--')
End Python code.
Program B:
Begin Python code:
def print_greatest(word):
print('--The Greatest *', end='')
print(word, end='')
print('* on Earth!--')
…
# Read from input: word = 'Show'
print_greatest(word)
…
…
# Read from input: word = Song
print_greatest(word)
End Python code.
Step 1: A print function can improve the organization of a program.
Step 2: The benefit of a print function increases with repeated calls. Both Program A and B output the formatted text twice, but the output code for Program B is only written once in the print_greatest() function. Program B uses two function calls print_greatest() to produce the same output as Program A.
Step 3: In Program B, any changes to the output, e.g. adding '!', have to be made only in the print_greatest() function. The print_greatest() function has one of its print statements changed from print('* on Earth--') to print('* on Earth!--').

Step 4: Without a function, like in Program A, output must be changed in multiple instances, which can be time-consuming and lead to errors. The two print statements print('* on Earth--') are both changed to print('* on Earth!--').

Animation captions:

1. A print function can improve the organization of a program.
2. The benefit of a print function increases with repeated calls. Both Program A and B output the formatted text twice, but the output code for Program B is only written once in the print_greatest() function.
3. In Program B, any changes to the output, e.g. adding '!', have to be made only in the print_greatest() function.
4. Without a function, like in Program A, output must be changed in multiple instances, which can be time-consuming and lead to errors.

PARTICIPATION ACTIVITY 6.2.4: Calling a print function multiple times.

Refer to the example programs above.

1) To output the phrase, "The Greatest [word] on Earth", with ten different words, what is the minimum number of times the main program would need to call print_greatest() in Program B?

   - 1
   - 10
   - 30

2) To output "in the Galaxy" instead of "on Earth" in the phrase (Ex: "--The Greatest *Cafe* in the Galaxy!-- "), how many statements need to be changed in Program A and Program B?

   - Program A: 1
     Program B: 1
   - Program A: 2
     Program B: 1
   - Program A: 2
     Program B: 2

Example: Menu system

Figure 6.2.1: Example: Menu System.

```
def print_menu():
    print("Today's Menu:")
    print('   1) Gumbo')
    print('   2) Jambalaya')
    print('   3) Quit\n')

quit_program = False

while not quit_program :
    print_menu()
    choice = int(input('Enter choice:
'))
    if choice == 3 :
        print('Goodbye')
        quit_program = True
    else :
        print('Order: ', end='')
        if choice == 1 :
            print('Gumbo')
        elif choice == 2 :
            print('Jambalaya')
        print()
```

```
Today's Menu:
   1) Gumbo
   2) Jambalaya
   3) Quit

Enter choice: 2
Order:
Jambalaya

Today's Menu:
   1) Gumbo
   2) Jambalaya
   3) Quit

Enter choice: 1
Order: Gumbo

Today's Menu:
   1) Gumbo
   2) Jambalaya
   3) Quit

Enter choice: 3
Goodbye
```


PARTICIPATION ACTIVITY 6.2.5: Example: Menu System.

Consider the example above.

1) How many times is print_menu() called?

- 1
- 2
- 3

2) Which of the following code statements if added to print_menu() would make the function no longer void?

- num_options = 3
- return 0
- return

CHALLENGE ACTIVITY 6.2.1: Enter the output of the print function.

566436.4601014.qx3zqy7

Start

Type the program's output


```
def print_age(user_age):
    print(f'He is {user_age}')

age_to_print = 23
print_age(age_to_print)
```

```
He is 23
```

1 2 3 4

Check Next

CHALLENGE ACTIVITY 6.2.2: Print functions.

566436.4601014.qx3zqy7

Start

Two food-cost pairs are read from input and are passed as arguments for print_item_price(). Define a f that has two parameters and outputs the following all on one line:

- 'One '
- the value of the first parameter
- ' costs '
- the value of the second parameter
- ' dollars.'

print_item_price() should not return any value.

▶ **Click here for example**


```
1
2 ''' Your code goes here '''
3
4 item_name1 = input()
5 item_price1 = int(input())
6 item_name2 = input()
7 item_price2 = int(input())
8
9 print_item_price(item_name1, item_price1)
10 print_item_price(item_name2, item_price2)
```

1 2 3

Check Next level

6.3 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'`.

PythonTutor: Polymorphic functions.

```
1 def add(x, y):
2     return x + y
3
4 print(f'add(5, 7) is {add(5, 7)}')
5 print(f"add('Tora', 'Bora') is {add('Tora', 'Bora')}"
```

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 9

Program output

Frames Objects

The function's behavior of adding 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 five 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 with 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"` declares a string variable in 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 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.

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 lower case, doing everything that a string can do, then treat the object like a string.

PARTICIPATION ACTIVITY 6.3.1: Dynamic and static typing.

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

6.4 Reasons for defining functions

Improving program readability

Programs can become hard for humans to read and understand. Decomposing a program into functions can aid program readability, yield an initially correct program, and ease 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 6.4.1: With user-defined functions, the main program is easy to understand.


```
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(f'Feet: {feet}')

calories = steps_to_calories(steps)
print(f'Calories: {calories}')
```

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

Figure 6.4.2: 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(f'Feet: {feet}')

minutes = steps / steps_per_minute
calories = minutes *
calories_per_minute_walking
print(f'Calories: {calories}')
```

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


PARTICIPATION ACTIVITY 6.4.1: 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

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 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 the 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 6.4.2: Redundant code can be replaced by multiple calls to one function.

```
pi_val = math.pi

pizza_diameter_1 = 12.0
circle_radius_1 = pizza_diameter_1 / 2.0
circle_area_1 = pi_val * circle_radius_1 *
             circle_radius_1

pizza_diameter_2 = 14.0
circle_radius_2 = pizza_diameter_2 / 2.0
circle_area_2 = pi_val * circle_radius_2 *
              circle_radius_2

total_pizza_area = circle_area_1 +
                 circle_area_2

print('A 12 and 14 inch pizza has', end=' ')
print(f'{total_pizza_area:.2f}',
  end=' ')
print('square inches combined.')
```

Main program with redundant code

```
def calc_circle_area(circle_diameter):
    pi_val = math.pi

    circle_radius = circle_diameter / 2.0
    circle_area = pi_val * circle_radius *
                circle_radius

    return circle_area


pizza_diameter_1 = 12.0
pizza_diameter_2 = 14.0

total_pizza_area =
  calc_circle_area(pizza_diameter_1) +
  calc_circle_area(pizza_diameter_2)

print('A 12 and 14 inch pizza has', end=' ')
print(f'{total_pizza_area:.2f}', end=' ')
print('square inches combined.')
```

Main program calls calc_circle_area() avoiding redundant code

Static figure: A code block is displayed.
Begin Python code:

```
pi_val = math.pi

pizza_diameter_1 = 12.0
circle_radius_1 = pizza_diameter_1 / 2.0
circle_area_1 = pi_val * circle_radius_1 * circle_radius_1

pizza_diameter_2 = 14.0
circle_radius_2 = pizza_diameter_2 / 2.0
circle_area_2 = pi_val * circle_radius_2 * circle_radius_2

total_pizza_area = circle_area_1 + circle_area_2

print('A 12 and 14 inch pizza has', end=' ')
print(f'{total_pizza_area:.2f}',
  end=' ')
print('square inches combined.')
```

End Python code.
Step 1: Circle area is calculated twice, leading to redundant code. The lines of code, circle_radius_1 = pizza_diameter_1 / 2.0, circle_area_1 = pi_val * circle_radius_1 * circle_radius_1, and circle_radius_2 = pizza_diameter_2 / 2.0, circle_area_2 = pi_val * circle_radius_2 * circle_radius_2, are highlighted.
Step 2: The redundant code can be replaced by defining a calc_circle_area() function. A new code block is displayed:
Begin Python code:

```
def calc_circle_area(circle_diameter):
    pi_val = math.pi

    circle_radius = circle_diameter / 2.0
    circle_area = pi_val * circle_radius * circle_radius

    return circle_area
```

End Python code.
Step 3: Then main program is simplified by calling the calc_circle_area() function from multiple places in the program. The remaining code is added to the bottom of the new code block:
Begin Python code:

```
def calc_circle_area(circle_diameter):
    pi_val = math.pi

    circle_radius = circle_diameter / 2.0
    circle_area = pi_val * circle_radius * circle_radius

    return circle_area
```

```
pizza_diameter_1 = 12.0
pizza_diameter_2 = 14.0

total_pizza_area = calc_circle_area(pizza_diameter_1) + calc_circle_area(pizza_diameter_2)

print('A 12 and 14 inch pizza has', end=' ')
print(f'{total_pizza_area:.2f}', end=' ')
print('square inches combined.')
```


End Python code.

The text, Main program with redundant code, appears below the original code block while the text, Main program calls calc_circle_area() avoiding redundant code, appears below the new code block.

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 6.4.3: 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

CHALLENGE ACTIVITY 6.4.1: Functions: Factoring out a unit-conversion calculation.

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(f'Miles: {miles_traveled:f}')
```

Sample output with inputs: 70.0 100.0

```
Miles: 116.666667
```

Learn how our autograder works

566436.4601014.qx3zqy7

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


Run

View your last submission

6.5 Writing mathematical functions

Mathematical functions

A function is defined as a mathematical calculation involving several numerical parameters and returning a numerical result. Ex: The following program uses a function to convert a person's height in feet and inches into centimeters.

PythonTutor: 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(f'Centimeters: {centimeters}')
```

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 12

Program output

Frames Objects

Most Americans only know their height in feet/inches, not in total inches. Human average height is increasing, attributed largely to better nutrition. (Source: Our World in Data: Human height.)

PARTICIPATION ACTIVITY 6.5.1: Mathematical functions.

Indicate which is a valid use of the height_US_to_cm() function above.

1) x = height_US_to_cm(5, 0)
   - Valid
   - Not valid

2) x = 2 * (height_US_to_cm(5, 0) + 1.0)
   - Valid
   - Not valid

3) x = (height_US_to_cm(5, 0) + height_US_to_cm(6, 1)) / 2.0
   - Valid
   - Not valid

4) Suppose pow(y, z) returns y to the power of z. Is the following valid? x = pow(2, pow(3, 2))
   - Valid
   - Not valid

zyDE 6.5.1: Temperature conversion.

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

Load default template...

```
1 def c_to_f():
2     # ETXMF
```

```
    return  # FIXME: Finish

temp_c = float(input('Enter temperature in Celsius: '))
temp_f = None

# FIXME: Call conversion function
# temp_f = ??

# FIXME: Print result
# print(f'Fahrenheit: {temp_f}')
```


50

Run

Calling functions in expressions

A function call 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 6.5.2: Function called twice in an expression.

```
def compute_square(num_to_square):
  return num_to_square * num_to_square


c2 = compute_square(7) + compute_square(9)

print(f'7 squared plus 9 squared is {c2}')
```

7 squared plus 9 squared is 130

Animation content:

Static figure: A code block and an output console are displayed.
Begin Python code:

```
def compute_square(num_to_square):
  return num_to_square * num_to_square


c2 = compute_square(7) + compute_square(9)

print(f'7 squared plus 9 squared is {c2}')
```

End Python code.

Step 1: compute_square() is called within an expression two times, first with the argument 7. 7 is passed to compute_square() and the value 49 is returned. The line of code, c2 = compute_square(7) + compute_square(9), is highlighted and the compute_square() function is called with the argument 7. The lines of code, def compute_square(num_to_square):, return num_to_square * num_to_square, are briefly highlighted and 49 is returned.
Step 2: compute_square() is called a second time with the argument 9, and the value 81 is returned to the expression. The lines of code, int ComputeSquare(int numToSquare) {, return num_to_square * num_to_square, are briefly highlighted and 81 is returned.
Step 3: The expression then evaluates to c2 = 49 + 81, which assigns variable c2 with 130. Lastly, the print statement executes. The line of code, print(f'7 squared plus 9 squared is {c2}'), is highlighted and the text, 7 squared plus 9 squared is 130, is displayed in the output console.

Animation captions:

1. compute_square() is called within an expression two times, first with the argument 7. 7 is passed to compute_square() and the value 49 is returned.
2. compute_square() is called a second time with the argument 9, and the value 81 is returned to the expression.
3. The expression then evaluates to c2 = 49 + 81, which assigns variable c2 with 130. Lastly, the print statement executes.

PARTICIPATION ACTIVITY 6.5.3: Function calls in an expression.

Given the following functions, 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)

- Valid
- Invalid

2) square_root(49.0) = z

- Valid
- Invalid

3) y = 1.0 + square_root(144.0)

- Valid
- Invalid

4) y = square_root(square_root(16.0))

- Valid
- Invalid

5) y = square_root()

- Valid
- Invalid

6) square_root(9.0)

- Valid
- Invalid

7) y = print_val(9.0)

- Valid
- Invalid

8) y = 1 + print_val(9.0)

- Valid
- Invalid

9) print_val(9.0)

- Valid
- Invalid

©zyBooks 11/07/24 14:53 2300507
Liz Vokac Main
KVCCCIS216JohnsonFall2024

CHALLENGE ACTIVITY 6.5.1: 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
```

Learn how our autograder works

566436.4601014.qx3zqy7

```
def find_max(num_1, num_2):
    max_val = 0.0

    if (num_1 > num_2):  # if num1 is greater than num2,
        max_val = num_1  # then num1 is the maxVal.
    else:                # Otherwise,
        max_val = num_2  # num2 is the maxVal
    return max_val

max_sum = 0.0

num_a = float(input())
num_b = float(input())
num_y = float(input())
num_z = float(input())

''' Your solution goes here '''
```

Run

View your last submission

Modular functions for mathematical expressions

Modularity allows more complex functions to incorporate simpler functions. Complex mathematical functions often call other mathematical functions. Ex: A function that calculates the volume or surface area of a cylinder calls a function that returns the area of the cylinder's base, which is needed for both calculations.

Figure 6.5.1: Program that calculates cylinder volume and surface area by calling a modular function for the cylinder's base.

```
import math

def calc_circular_base_area(radius):
   return math.pi * radius * radius

def calc_cylinder_volume(baseRadius, height):
   return calc_circular_base_area(baseRadius) * height

def calc_cylinder_surface_area(baseRadius, height):
   return (2 * math.pi * baseRadius * height) + (2 *
calc_circular_base_area(baseRadius))

radius = float(input('Enter base radius: '))
height = float(input('Enter height: '))

print(f'Cylinder volume: {calc_cylinder_volume(radius,
height):.3f}')
print(f'Cylinder surface area:
{calc_cylinder_surface_area(radius, height):.3f}')
```

```
Enter base
radius: 10
Enter height: 5
Cylinder volume:
1570.796
Cylinder surface
area: 942.478
```


CHALLENGE ACTIVITY 6.5.2: Writing mathematical functions.

566436.4601014.qx3zqy7

Start

Complete the function convert_to_pounds() that has one parameter as a mass in kilograms. The functi converted to pounds, given that 1 kilogram = 2.20462 pounds.

Ex: If the input is `10`, then the output is:

```
Result: 22.046 pounds
```

```
1  KG_TO_LBS = 2.20462
2
3  def convert_to_pounds(user_kilograms):
4
5      ''' Your code goes here '''
6
7  kilograms = int(input())
8
9  # Print with value rounded to 3 decimal places
10 print(f'Result: {convert_to_pounds(kilograms):.3f} pounds')
```

1 2

Check Next level

6.6 Function stubs

Incremental development and function stubs

Programs are 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 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.

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

```
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(f'Feet: {feet}')

calories = steps_to_calories(steps)
print(f'Calories: {calories}')
```

```
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
```


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.

```
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(f'Perimeter of triangle:
{get_perimeter_length(coordinates)}')
```



```
Traceback (most recent
call last):
 File "<stdin>", line 10,
in glt;module<
 File "<stdin>", line 2,
in get_points
NotImplementedError
```

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 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

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
```

Learn how our autograder works

566436.4601014.qx3zqy7

```
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(f'Avg: {avg_result}')
```


Run

View your last submission

6.7 Functions with branches/loops

Example: Auction website fee calculator

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 fee charged by eBay when a user sells an item online.

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

```
def calc_ebay_fee(sell_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 sell_price <= 50:
        fee  = fee + (sell_price*p50)
    elif sell_price <= 1000:
        fee = fee + (50*p50) + ((sell_price-
50)*p50_to_1000)
    else:
        fee = fee + (50*p50) + ((1000-
50)*p50_to_1000) \
                  + ((sell_price-1000)*p1000)

    return fee

selling_price = float(input('Enter item selling
price (ex: 65.00): '))
print(f'eBay fee:
${calc_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 calc_ebay_fee(), how many assignment statements will execute?

Check Show answer

2) What does calc_ebay_fee() return if the sell_price argument is 0.0 (show your answer in the form #.#)?

Check Show answer

3) What does calc_ebay_fee() return if the sell_price argument is 100.0 (show your answer in the form #.#)?

Check Show answer

4) Write a function call using the calc_ebay_fee() function to determine the fee for a selling price of 15.23, and store the result in a variable named my_fee.

Check Show answer

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...

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

5 99 -44 0 12

Run

PARTICIPATION ACTIVITY 6.7.2: Analyzing the numbers program.

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

Check Show answer

CHALLENGE ACTIVITY 6.7.1: Output of functions with branches/loops.

566436.4601014.qx3zqy7

Start

Type the program's output

```
def normalize_grade(grade):
   upper_bound = 74
   lower_bound = 53

   if grade > upper_bound:
      return upper_bound
   elif grade <= lower_bound:
      return lower_bound
   else:
      return grade

grade1 = normalize_grade(19)
grade2 = normalize_grade(82)
grade3 = normalize_grade(69)

print(grade1)
print(grade2)
print(grade3)
```

```
53
74
69
```

1 2 3

Check Next

CHALLENGE ACTIVITY 6.7.2: 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 * bag_ounces followed by " seconds".

Sample output with input: 7

```
42 seconds
```

Learn how our autograder works

566436.4601014.qx3zqy7

```
''' Your solution goes here '''

bag_ounces = int(input())
print_popcorn_time(bag_ounces)
```


Run

View your last submission

CHALLENGE ACTIVITY | 6.7.3: Functions with branches/loops.

566436.4601014.qx3zqy7

Start

Define a function calculate_priority() that takes one parameter as the number of project tasks to be cor project's priority. The project's priority is returned as follows:

- If the task's count is more than 95, then the priority is 3.
- If the task's count is more than 20 and up to 95, then the priority is 2.
- Otherwise, the priority is 1.

▶ **Click here for examples**

```
''' Your code goes here '''

input_tasks_num = int(input())

print(f'Testing 100: {calculate_priority(100)}')
print(f'Testing user input: {calculate_priority(input_tasks_num)}')
```

1 2

Check Next level

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 bytecode 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.

Figure 6.8.1: Python bytecode.

Program	Bytecode

`def add_one(x):`
`  y = x + 1`
`  return y`
	`0 LOAD_FAST 0 (x)`
	`3 LOAD_CONST 1 (1)`
	`6 BINARY_ADD`
	`7 STORE_FAST 1 (y)`
	`10 LOAD_FAST 1 (y)`
	`13 RETURN_VALUE`

All Python code is compiled before execution 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.

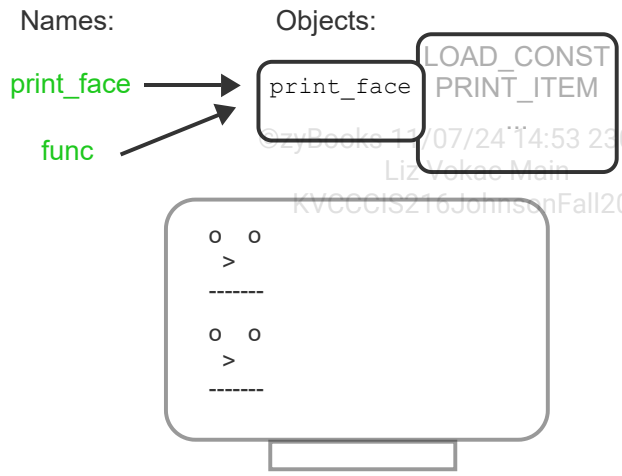
Because a function is an object, a function can be used in an assignment statement just like other objects. This is illustrated in the following animation.

PARTICIPATION ACTIVITY 6.8.1: Functions are objects.

```
def print_face():
    # print face statements...


print_face()

func = print_face
func()
```



Animation content:

Static figure: A code block, an output console, and the text, Names:, and Objects:, are displayed.
Begin Python code:

```
def print_face():
  # print face statements...


print_face()

func = print_face
func()
```

End Python code.

Step 1: def print_face() creates a new function object. The line of code, def print_face():,is highlighted, and the text, print_face, appears under Names: with an arrow pointing to the text, print_face, which also appears under Objects:.
Step 2: The compiled bytecode of print_face function is stored in the function object. The line of code, # print face statements..., is highlighted, and the text, LOAD_CONST PRINT_ITEM..., appears next to print_face under Objects.
Step 3: When print_face() is called, the print_face() function runs. The line of code, print_face(), is briefly highlighted. The line of code, # print face statements..., is highlighted, and the text, LOAD_CONST PRINT_ITEM..., is highlighted and a face is printed in the output console. The output console now contains the text:

```
o   o
 >
-------
```

Step 4: The line of code, func = print_face, is highlighted, and the text, func, appears under Names: with an arrow pointing to the text, print_face, under Objects:. The line of code, func(), is briefly highlighted. The line of code, # print face statements..., is highlighted, and the text, LOAD_CONST PRINT_ITEM..., is highlighted and a face is printed in the output console. The output console now contains the text:

```
o   o
 >
-------
o   o
 >
-------
```


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().

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: print_human_head() and print_monkey_head(). A third function, print_figure(), accepts a function as an argument, calling that function to print a head and then print a body.

Figure 6.8.2: Functions can be passed as arguments.

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

def print_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(print_monkey_head)
elif choice == 2:
    print_figure(print_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
     |
   --|--
  /  |  \
@    |    @
     |
    /|\
   @   @
```


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 print_human_head() or print_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 only support the call operation.

Function attributes

Functions also support adding attributes with the attribute reference "." operator, but that

concept is out of scope for the discussion here.

PARTICIPATION ACTIVITY 6.8.2: Function objects.

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 'meow':

```
def print_cat():
    print('meow')

def print_pig():
    print('oink')

print_cat = print_pig
print_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

6.9 Functions: Common errors

Copy-paste errors

A common error is to copy and paste code among functions but 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
```

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 correct code. When you copy and paste code, be extremely vigilant in making all necessary modifications. Just as dark alleys or wet roads may be dangerous and cause you to observe your surroundings or drive carefully, the awareness of error potential in copying and pasting may cause you to modify a pasted function correctly.

PARTICIPATION ACTIVITY 6.9.1: Copied and 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 copied and pasted code below.

1)

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

return sum

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.

PythonTutor: 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(f'Distance walked in feet: {feet}')
```


line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 9

Program output

Frames Objects

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 6.9.2: 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 returning the square of an integer?

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

- Yes
- No

5) Is this function correct for returning the square of 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
```

Learn how our autograder works

566436.4601014.qx3zqy7


```
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(f'{value_c} C is {celsius_to_kelvin(value_c)} K')
11
```

```
value_k = float(input())
print(f'{value_k} K is {kelvin_to_celsius(value_k)} C')
```

Run

View your last submission

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`. 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***.

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(f'Centimeters: {height_US_to_centimeters(feet, inches)}')
```


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

Global variables

In contrast, a variable defined outside of a function is called a ***global variable***. A global variable's scope extends from the 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.

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

```
employee_name = 'N/A'

def get_name():

    name = input('Enter employee
name:')
    employee_name = name

get_name()
print(f'Employee name:
{employee_name}')
```

```
Enter employee name: Romeo Montague
Employee name: N/A
```

```
employee_name = 'N/A'

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

get_name()
print(f'Employee name:
{employee_name}')
```



```
Enter employee name: Juliet Capulet
Employee name: Juliet Capulet
```

The global statement must be applied to any global variable that is assigned 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 naming can be 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 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 call a function, the interpreter must have 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.

```
employee_name = 'N/A'

get_name()
print(f'Employee name:
{employee_name}')

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

```
NameError: name 'get_name' is not
defined
```

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.
   - ○ True
   - ○ False

2) A local variable's scope extends from a function definition's ending colon ":" to

the end of the function.

- True
- False

3) A global statement must be used to assign a global variable inside a function.

- True
- False

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

- True
- False

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.

```
x = 5
y = 100
z = x + y
```

Global namespace

x	5
y	100
z	105

Global variables are tracked in the global namespace

Animation content:

Static figure: Begin Python code:
x = 5
y = 100
z = x + y
End Python code.
A table with the label Global namespace is also shown.
Step 1: Global variables are tracked in the global namespace. The Python interpreter reads the line x = 5 in the program. x and 5 are entered in the first row of the Global namespace table. Then, the Python interpreter does the same with the next line of the program, reading y = 100 and entering y and 100 in the second row of the table. The table is as follows:
Global namespace

x	5
y	100

Step 2: The value of variables is found by looking in the namespace. The Python interpreter reads the third line of the program, z = x + y. The interpreter locates x in the Global namespace table and finds that the value of x is 5. Then, the interpreter locates y in the Global namespace table, and finds that the value of y is 100. Using these values, the value of x + y is calculated to be 105. z and 105 are entered in the third row of the Global namespace table. The table is as follows:
Global namespace

x	5
y	100
z	105

Animation captions:

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

A namespace is 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_var': 'This is a variable', 'my_func':
<function my_func at 0x2349d4>}
```


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.

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

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 – 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***.

Levels of scope

In fact four levels of scope exist. A level between the local function scope and global scope was omitted for clarity. A function can be defined within another function – in such a case the scope of the outer function is checked before the global scope is checked.

PARTICIPATION ACTIVITY 6.11.2: Scope resolution.

```
daily_cals = 2300  # Daily calories
soda_cals = 200

def drink_soda(cals_left):
    return cals_left - soda_cals

daily_cals = drink_soda(daily_cals)
```

Global namespace

daily_cals	2100
soda_cals	200
drink_soda	<funct>

Built-in namespace

int()	<funct>
...	...
input()	<funct>

Global variables are tracked in the global namespace.

Animation content:

Static figure: Begin Python code:
daily_cals = 2300 # Daily calories
soda_cals = 200

def drink_soda(cals_left):
  return cals_left - soda_cals

daily_cals = drink_soda(daily_cals)
End Python code.
A blank table with the label Global namespace and a table with the label Built-in namespace are also shown. The Built-in namespace table is as follows:

int()	open angle bracket funct close angle bracket
...	...

input()	open angle bracket funct close angle bracket

The ellipses in the second row indicate that there are other rows in this table that are not shown.
Step 1: Global variables are added to the global namespace. The Python interpreter reads the line daily_cals = 2300 and enters daily_cals and 2300 in the global namespace table. Then the interpreter reads soda_cals = 200 and enters soda_cals and 200 in the table. Then the interpreter reads the function definition def drink_soda(cals_left):. The interpreter enters drink_soda and open angle bracket funct close angle bracket in the global namespace table. The global namespace table is as follows:

daily_cals	2300
soda_cals	200
drink_soda	open angle bracket funct close angle bracket

Step 2: Calling a function creates a new local namespace for local variables. The Python interpreter reads the line daily_cals = drink_soda(daily_cals). The interpreter locates drink_soda in the Global namespace table and finds that it is a function. The interpreter also locates daily_cals in the Global namespace table and finds that its value is 2300. The interpreter finds the header of the function definition for drink_soda in the program, which reads def drink_soda(cals_left):. Since the initial function call for drink_soda had the parameter daily_cals, cals_left in the function definition header is substituted with the value of daily_cals, which is 2300. Then, a new namespace table is created, with the heading Local namespace. cals_left is entered in the Local namespace table with the value 2300.
Step 3: Calling a function creates a new local namespace for local variables. The interpreter reads the line return cals_left - soda_cals. cals_left is located in the Local namespace table, and its value is 2300. soda_cals is not found in the Local namespace table, but is found in the Global namespace table. The value of soda_cals is 200. The value of cals_left = soda_cals is calculated. It is 2300 - 200, which is 2100.
Step 4: The local namespace is removed when the function returns. The Python interpreter returns to the line call daily_cals = drink_soda(daily_cals), and substitutes the function call with the calculated value 2100. The local namespace table disappears. daily_cals is located in the Global namespace table, and its value is updated to 2100.

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.

If unable to drag and drop, refresh the page.

Scope **Namespace** **locals()** **Scope resolution**

	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

PARTICIPATION ACTIVITY 6.11.4: Namespaces.

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(f'Player: {str(player_roll)}    Troll: {str(troll_roll)}')
```


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 explains 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 f'Sum: {tmp:f}'` 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".

PythonTutor: Function scope.

```
1  def avg(a, b):
2      tmp = (a + b) / 2.0  # Creates tmp in local namesp
3      return tmp
4
5  a = 5
6  b = 10
7  tmp = a + b  # Creates tmp in global namespace
8
9  print(f'Avg: {avg(a, b):f}')
10 print(f'Sum: {tmp:f}')
```


line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 10

Program output

Frames Objects

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 6.11.5: 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

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.

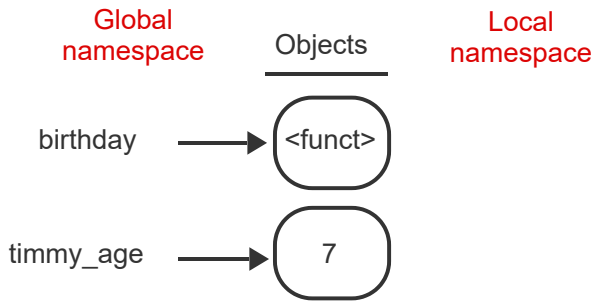
PARTICIPATION ACTIVITY 6.12.1: Assignments to parameters have no effect outside the function.

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

timmy_age = 7

birthday(timmy_age)
print(f'Timmy is {timmy_age}')
```

Timmy is 7



Assigning age with a new value in the function doesn't change timmy_age.

Animation content:

Static figure:
Begin Python code:

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

timmy_age = 7

birthday(timmy_age)
print(f'Timmy is {timmy_age}')
```

End Python code.
Also shown are a program output box and a diagram showing objects from the program. The diagram indicates that objects can be referenced in the global namespace, in a local namespace, or in both.
Step 1: timmy_age and age reference the same object. The Python interpreter reads the first line of the program, def birthday(age): and stores a function object. The variable name birthday in the global namespace points to the function object. Then the interpreter reads the line timmy_age = 7. The interpreter stores an object with value 7, and the variable name timmy_age points to the 7 object in the global namespace. The interpreter reads the function call birthday(timmy_age). The interpreter goes to the function definition and reads the line def birthday(age): again. In the function call, the object given for the age parameter is called timmy_age. In the objects diagram, the name age is added for the object with value 7 in the local namespace. Now timmy_age in the global namespace and age in the local namespace both point to the object with value 7.
Step 2: Assigning the parameter age with a new value doesn't change timmy_age. The interpreter enters the function and reads the line age = age + 1. A new object with value 8 is created, and the pointer for age is changed. Now the variable name age in the local namespace points to the object with value 8. The name timmy_age in the global namespace still points to the object with value 7.
Step 3: Since timmy_age has not changed, "Timmy is 7" is displayed. The Python interpreter exits the function, and the object with value 8 and the local namespace name age are deleted. The interpreter reads the line print(f'Timmy is {timmy_age}'). In the program output box, "Timmy is 7" is output.

Animation captions:

1. timmy_age and age reference the same object.
2. Assigning the parameter age with a new value doesn't change timmy_age.
3. Since timmy_age has not changed, "Timmy is 7" is displayed.

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 is 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.

PythonTutor: Modification of a list inside a function.

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

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 7

Program output

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(num_list[:])`.

PythonTutor: Modification of a list inside a function.

```
1 def modify(num_list):
2     num_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!
```


line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 7

Program output

Frames Objects

zyDE 6.12.1: List argument modification.

Address the FIXME comments. Move the respective code from the while-loop to the created 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 in 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
```

```
1
Johnny B+
1
```

Run

PARTICIPATION ACTIVITY 6.12.2: 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 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

CHALLENGE ACTIVITY | 6.12.1: Function arguments.

566436.4601014.qx3zqy7

Start

item_list is read from input. shift_right() has one parameter list_to_modify, and shifts list_to_modify righ
copy of the list item_list as the argument to avoid modifying item_list.

▶ **Click here for example**

```
def shift_right(list_to_modify):
    index = len(list_to_modify) - 1
    temp = list_to_modify[len(list_to_modify) - 1]
    while index > 0:
        list_to_modify[index] = list_to_modify[index - 1]
        index -= 1
    list_to_modify[0] = temp
    print(f'Shifted right: {list_to_modify}')

item_list = input().split()

''' Your code goes here '''

print(f'Original: {item_list}')
```

1 2

Check Next level

6.13 Keyword arguments and default parameter values

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 order.

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 four 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

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(f'{month}/{day}/{year}')
    elif style == 1:  # European
        print(f'{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 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

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 four people. Use the default tax percentage and tip amount.

Check Show answer

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

Check Show answer

CHALLENGE ACTIVITY 6.13.1: Keyword arguments and default parameters.

566436.4601014.qx3zqy7

Start

Type the program's output

```
def show(a, b, c):
    print(f'{a}\{b}\{c}')

show(b=4, a=5, c=7)
```

5\4\7

1 2 3

Check Next

CHALLENGE ACTIVITY 6.13.2: Return number of pennies in total.

Given that 1 dollar = 100 pennies, write a function number_of_pennies() that takes two arguments: the number of dollars, and an optional number of pennies. number_of_pennies() should return the total number of pennies.

The code reads three values from input and calls number_of_pennies() twice.

Ex: If the input is 5 6 4, number_of_pennies() is first called with arguments 5 and 6, representing $5.06, and then number_of_pennies() is only called with argument 4, representing $4.00.

Then the output is:


```
506
400
```

Learn how our autograder works

566436.4601014.qx3zqy7

```
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 your last submission

CHALLENGE ACTIVITY 6.13.3: 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 four 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`. Assume that the tip is calculated from the amount of the bill before tax.

Sample output with inputs: 25 2

```
Cost per diner: $15.50
```


Sample output with inputs: 100 2 0.075 0.21

```
Cost per diner: $64.25
```

Learn how our autograder works

566436.4601014.qx3zqy7

```
# FIXME: Write the split_check function. HINT: Calculate the amount of tip and tax
# add to the bill total, then divide by the number of diners.

''' Your solution goes here '''

bill = float(input())
people = int(input())

# Cost per diner at the default tax and tip percentages
print(f'Cost per diner: ${split_check(bill, people):.2f}')

bill = float(input())
people = int(input())
new_tax_percentage = float(input())
new_tip_percentage = float(input())

# Cost per diner at different tax and tip percentages
```

Run

View your last submission

6.14 Arbitrary argument lists

Arbitrary arguments

Sometimes a programmer doesn't know how many arguments a function requires. A function definition can include an ***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 print_sandwich(bread, meat, *args):
    print(f'{meat} on {bread}', end=' ')
    if len(args) > 0:
        print('with', end=' ')
    for extra in args:
        print(extra, end=' ')
    print('')

print_sandwich('sourdough', 'turkey', 'mayo')
print_sandwich('wheat', 'ham', 'mustard',
'tomato', 'lettuce')
```



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

Adding a final function parameter of ****kwargs***, short for ***keyword arguments***, creates a dictionary containing "extra" arguments not defined in the function definition. 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 print_sandwich(meat, bread, **kwargs):
    print(f'{meat} on {bread}')
    for category, extra in kwargs.items():
        print(f'   {category}: {extra}')
    print()

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

```
turkey on
sourdough
   sauce: mayo

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


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.

PythonTutor: 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 += f' --{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
```

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 19

Program output

Frames Objects

PARTICIPATION ACTIVITY 6.14.1: 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
```

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

print_stats(
)
```

Check Show answer

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 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(f'Average score: {average}')
print(f'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]`.

Unpacking is an operation that allows a statement to perform multiple assignments at once to variables in a tuple or list. Ex: The statement `average, standard_deviation = get_grade_stats(student_scores)`, uses 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

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 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.

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

```
1  def ticket_price(origin, c
2      """Calculates the pric
3      Only one of coach or f
4
5      Arguments:
6      origin -- string repre
7      destination -- string
8
9      Optional keyword argum
10     coach -- Boolean. True
11     first_class -- Boolear
12
13     """
14     #Function body stateme
15
16 help(ticket_price)
17
```

The `help()` function works with most of the built-in Python names, since the language creators provided 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.

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.

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

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


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

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 the initial position of 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 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...

```
1
2  import math
3
4  def trajectory(t, a, v, g,
5      """Calculates new x,y
6      x = v * t * math.cos(a
7      y = h + v * t * math.s
8      return (x,y)
9
10 def degree_to_radians(degr
11     """Converts degrees to
12     return ((degrees * mat
13
14 gravity = 9.81 # Earth gra
15 time = 1.0 # time (s)
16 x_loc = 0
17
```

```
45
100
3
```

Run

PARTICIPATION ACTIVITY 6.17.2: Projective location.

Questions refer to the function trajectory above.

1) trajectory() cannot return two values (for x and y), so trajectory() 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 y_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

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
```

Learn how our autograder works

566436.4601014.qx3zqy7

```
gas_const = 8.3144621

''' Your solution goes here '''

gas_pressure = float(input())
gas_moles = float(input())
gas_temperature = float(input())
gas_volume = 0.0

gas_volume = compute_gas_volume(gas_pressure, gas_temperature, gas_moles)
print(f'Gas volume: {gas_volume} m^3')
```

Run

View your last submission

6.18 Lab training: Unit tests to evaluate your program

Auto-graded programming assignments may use a *Unit test* to test small parts of a program. Unlike a *Compare output test*, which evaluates your program's output for specific input values, a *Unit test* evaluates individual functions to determines if each function:

- is named correctly and has the correct parameters and return type
- calculates and returns the correct value (or prints the correct output)

The zyLabs auto-grader runs **main.py** as a script. In **main.py**, the line `if __name__ == '__main__':` is used to separate the main code from the functions' code so that each function can be unit tested. Enter statements to be run as the main code under `if __name__ == '__main__':`. Indent the statements so the statements belong to the if block. Refer to the subsection *Importing modules and executing scripts* under section *Module basics* for more information about running a program as a script.

Note: Do not remove `if __name__ == '__main__':` from the code. Otherwise, the unit tests will fail even though the program produces the correct output.

This example lab uses multiple unit tests to test the kilo_to_pounds() function.

Complete a program that takes a weight in kilograms as input, converts the weight to pounds, and then outputs the weight in pounds. 1 kilogram = 2.204 pounds (lbs).

The program must define the following function:
`def kilo_to_pounds(kilos)` - take `kilos` as a parameter, convert `kilos` from kilograms to pounds, and return the weight in pounds.

Ex: If the input of the program is:


```
10
```

10 is passed to kilo_to_pounds() and the output of the program is:

```
22.040 lbs
```

The program below has an error in the kilo_to_pounds() function.

1. Try submitting the program for grading (click "Submit mode", then "Submit for grading"). Notice that the first two test cases fail, but the third test case passes. The first test case fails because the program outputs the result from the kilo_to_pounds() function, which has an error. The second test case uses a *Unit test* to test the kilo_to_pounds() function, which fails.
2. Change the kilo_to_pounds() function to multiply the variable kilos by 2.204, instead of dividing. The return statement should be: `return (kilos * 2.204)`. Submit again. Now the test cases should all pass.

Note: A common error is to mistype a function name with the incorrect capitalization. Function names are case sensitive, so if a lab program asks for a kilo_to_pounds() function, a kilo_To_Pounds() function that works for you in "Develop mode" will result in a failed unit test. The unit test will not be able to find kilo_to_pounds().

LAB ACTIVITY | 6.18.1: Lab training: Unit tests to evaluate your program | 3 / 3

main.py

```
1 Loading latest submission...
```

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

Program output displayed here

Coding trail of your work What is this?

Retrieving signature

6.19 LAB: Driving costs - functions

Write a function driving_cost() with parameters miles_per_gallon, dollars_per_gallon, and miles_driven, that returns the dollar cost to drive those miles. All items are of type float. The function called with arguments (20.0, 3.1599, 50.0) returns 7.89975.

The main program's inputs are the car's miles per gallon and the price of gas in dollars per gallon (both float). Output the gas cost for 10 miles, 50 miles, and 400 miles, by calling your driving_cost() function three times.

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

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

Ex: If the input is:

```
20.0
3.1599
```

the output is:

```
1.58
7.90
63.20
```


Your program must define and call a function:

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

566436.4601014.qx3zqy7

LAB ACTIVITY | 6.19.1: LAB: Driving costs - functions 10 / 10

main.py

```
1 Loading latest submission...
```


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)

```
37
3.30
```

Run program

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

Program output displayed here

Coding trail of your work [What is this?](#)

Retrieving signature

6.20 LAB: Step counter

This section's content is not available for print.

6.21 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
```

The program must define and call the following two functions. Define a function named int_to_reverse_binary() that takes an integer as a parameter and returns a string of 1's and 0's representing the integer in binary (in reverse). Define a function named string_reverse() that takes an input string as a parameter and returns a string representing the input string in reverse.

```
def int_to_reverse_binary(integer_value)
def string_reverse(input_string)
```


LAB ACTIVITY 6.21.1: LAB: Convert to binary - functions 7 / 10

main.py

```
1 Loading latest submission...
```

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)

122

Run program

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

Program output displayed here

Coding trail of your work What is this?

Retrieving signature

6.22 LAB: Swapping variables

This section's content is not available for print.