Lecture 6: Functions

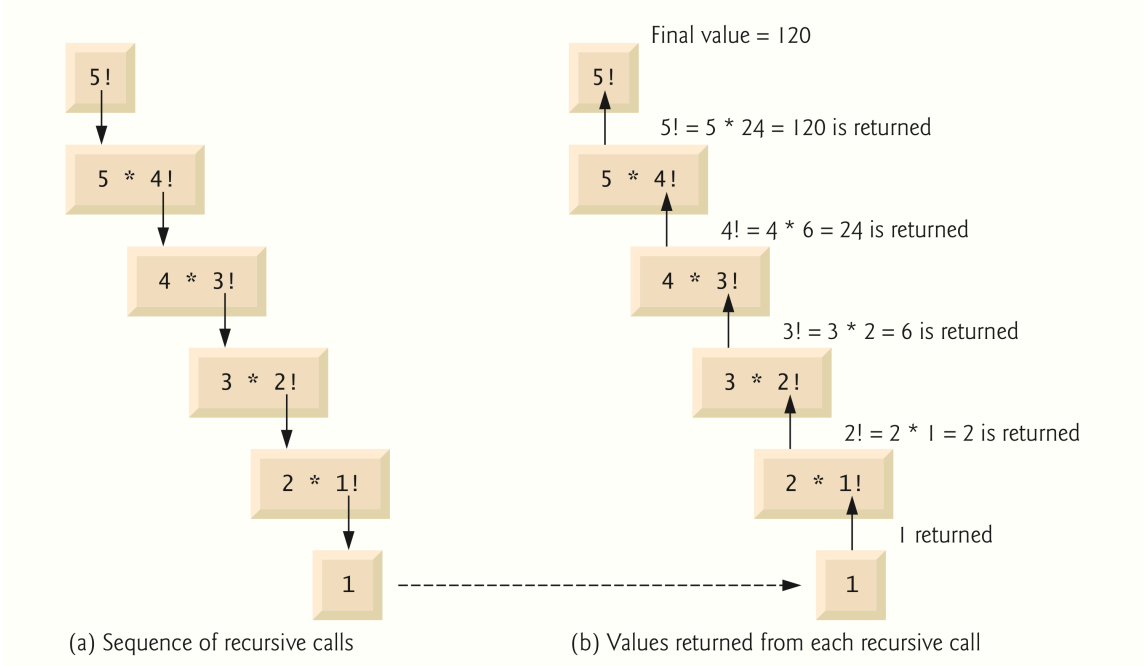
INF 605 · 2026-09-15

Open in Colab → **File → Save a copy in Drive**. You cannot edit the class original.

Start-of-class quiz

Type answers in the Canvas attendance quiz (empty boxes only). The questions stay here.

1. Last class was **Lists**. How do you get the last item of `xs` without hard-coding the length?
2. What does `xs.append(3)` do to `xs`?



(a) Sequence of recursive calls (b) Values returned from each recursive call

4.2 Defining Functions

- `square` function that calculates the square of its argument.

In [1]:
```python
def square(number):
    """Calculate the square of number."""
    return number ** 2
```

The next cell runs `square(7)`.

```
In [2]: square(7)
```

```
Out[2]: 49
```

The next cell runs `square(2.5)`.

```
In [3]: square(2.5)
```

```
Out[3]: 6.25
```

- Calling `square` with a non-numeric argument like `'hello'` causes a `TypeError` because the exponentiation operator (`**`) works only with numeric values

```
In [4]: square("Clark")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 square("Clark")

Cell In[1], line 3, in square(number)
      1 def square(number):
      2     """Calculate the square of number."""
----> 3     return number ** 2

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Defining a Custom Function

- Definition begins with the (`def` **keyword**, followed by the function name, a set of parentheses and a colon (`:`).
- By convention function names should begin with a lowercase letter and in multiword names underscores should separate each word.
- Required parentheses contain the function's **parameter list**.
- Empty parentheses mean no parameters.
- The indented lines after the colon (`:`) are the function's **block**
  - Consists of an optional docstring followed by the statements that perform the function's task.

Explanation:

- `def greet()`: `greet` is the function's name, and there are no parameters inside the parentheses.
- `print("Hello, welcome to learning Python!")`: The function prints a message.
- When you call `greet()`, the function prints the greeting.

In [5]:
```python
def greet():
    """This function prints a greeting message."""
    print("Hello, welcome to learning Python!")

# Call the function
greet()
```

```
Hello, welcome to learning Python!
```

Explanation:

- `name` : This is a parameter that the function expects when called.
- The function uses the name provided when it's called to greet the person.

In [6]:
```python
def greet_person(name):
    """This function greets the person by their name."""
    print(name)
    print(f"Hello, {name}, welcome to Python!")

# Call the function with an argument
greet_person("Mike")
```

```
Mike
Hello, Mike, welcome to Python!
```

Explanation:

- The function takes two parameters `a` and `b` and returns their sum using the `return` statement.
- You can store the result of the function in a variable and use it later, as shown in the example.

In [7]:
```python
def add_numbers(a, b):
    """This function returns the sum of two numbers."""
    return a + b

# Call the function and store the result
result = add_numbers(5, 3)
print(f"The sum is: {result}")
```

```
The sum is: 8
```

Explanation:

- In this case, the function has a default value for the parameter `name`, so if no argument is passed, it will greet with "Guest."

In [8]:
```python
def greet_person(name="Guest"):
    """This function greets the person, using 'Guest' as default if no name
    print(f"Hello, {name}, welcome!")

# Call the function with and without an argument
```

```
greet_person("Ron")
greet_person()  # Uses the default value 'Guest'
```

```
Hello, Ron, welcome!
Hello, Guest, welcome!
```

Specifying a Custom Function's Docstring

- *Style Guide for Python Code*: First line in a function's block should be a docstring that briefly explains the function's purpose.

Returning a Result to a Function's Caller

- Function calls also can be embedded in expressions:

In [9]:
```
print('The square of 7 is', square(7))
```

```
The square of 7 is 49
```

- Three Ways to Return a Result to a Function's Caller
  - `return` followed by an expression.
  - `return` without an expression implicitly returns `None` —represents the **absence of a value** and **evaluates to `False` in conditions**.
  - **No `return` statement implicitly returns `None`**.

In [10]:
```
# 1. Using 'return' with an expression
def add_numbers(a, b):
    """This function returns the sum of two numbers."""
    return a + b

result1 = add_numbers(3, 4)  # This will return 7
print(f"Result 1: {result1}")

# 2. Using 'return' without an expression (implicitly returns None)
def greet():
    """This function prints a greeting and returns None."""
    print("Hello!")
    return

result2 = greet()  # This returns None
print(f"Result 2: {result2}")  # Output will be 'None'

# 3. No 'return' statement (implicitly returns None)
def say_goodbye():
    """This function prints a goodbye message but has no return statement."""
    print("Goodbye!")

result3 = say_goodbye()  # This also returns None
print(f"Result 3: {result3}")  # Output will be 'None'
```

```
Result 1: 7
Hello!
Result 2: None
Goodbye!
Result 3: None
```

What Happens When You Call a Function

- Parameters exist only during the function call.
- Created on each call to the function to receive arguments.
- Destroyed when the function returns its result to the caller.
- A function's parameters and variables defined in its block are all **local variables**.

In [11]:
```python
def multiply(x, y):
    """This function multiplies two numbers and prints the result."""
    result = x * y  # 'result' is a local variable
    print(f"Multiplying {x} and {y} gives {result}")
    return result  # The result is returned to the caller

# Calling the function with arguments
product = multiply(5, 3)

# This line will print the result of the multiplication
print(f"The product is: {product}")

# Uncommenting the following lines will cause an error because 'x', 'y', and
print(x)
# print(result)
```

```
Multiplying 5 and 3 gives 15
The product is: 15
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[11], line 14
     11 print(f"The product is: {product}")
     13 # Uncommenting the following lines will cause an error because 'x',
'y', and 'result' only exist inside the function
---> 14 print(x)
     15 # print(result)

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

Accessing a Function's Docstring Via IPython's Help Mechanism

- Following a function's name with `?` in IPython displays its docstring:

In [12]:
```python
square?
```

- If the function's source code is accessible from IPython, `??` displays the function's docstring and full source-code definition:

In [13]:
```
square??
```

Exercise (celsius)

A weather app stores temperatures in Celsius. Write `celsius_to_fahrenheit(c)` that returns the Fahrenheit value. Use a return statement. Call it on `0` and on `100`.

In [ ]:
```
def celsius_to_fahrenheit(c):
    # convert Celsius to Fahrenheit
    return None

print(celsius_to_fahrenheit(0))
print(celsius_to_fahrenheit(100))
```

Solution

In [1]:
```
def celsius_to_fahrenheit(c):
    return (9 / 5) * c + 32

print(celsius_to_fahrenheit(0))
print(celsius_to_fahrenheit(100))
```

```
32.0
212.0
```

4.3 Functions with Multiple Parameters

- `maximum` function that determines and returns the largest of three values.

In [15]:
```
def maximum(value1, value2, value3):
    """Return the maximum of three values."""
    max_value = value1
    if value2 > max_value:
        max_value = value2
    if value3 > max_value:
        max_value = value3
    return max_value
```

The next cell runs `maximum(12, 27, 36)`.

In [16]:
```
maximum(12, 27, 36)
```

Out[16]: 36

The next cell runs `maximum(12.3, 45.6, 9.7)`.

In [17]:
```
maximum(12.3, 45.6, 9.7)
```

Out[17]: 45.6

The next cell runs `maximum('yellow', 'red', 'orange')`.

In [18]:
```
maximum('yellow', 'red', 'orange')
```

Out[18]: 'yellow'

In-class exercises: find as many rules as possible for string comparison.

In [19]:
```
'A' < 'a'
```

Out[19]: True

The next cell runs `'b' > 'B'`.

In [20]:
```
'b' > 'B'
```

Out[20]: True

The next cell runs `'ab' < 'ba'`.

In [21]:
```
'ab' < 'ba'
```

Out[21]: True

The next cell runs `'c' < 'd'`.

In [22]:
```
'c' < 'd'
```

Out[22]: True

In [ ]:
```

```

In [ ]:
```

```

In [23]:
```
# 1. Alphabetical Order: Python compares strings based on alphabetical order
# It compares the first character of each string, moving to the next charact
result_alpha = max('banana', 'apple', 'cherry')
print("The largest alphabetically is:", result_alpha)
# 'cherry' is the largest because 'c' comes after 'b' and 'a' in alphabetica


# 2. Case Sensitivity: Uppercase letters are considered smaller than lowerca
result_case = max('apple', 'Apple')
print("The largest considering case sensitivity is:", result_case)
# 'apple' is larger because lowercase 'a' comes after uppercase 'A'.
```

```
# 3. Numbers as Strings: Python compares strings character by character, eve
result_num = max('123', '45', '67')
print("The largest number as a string is:", result_num)
# '67' is the largest because Python compares character by character:
# '1' vs '4', '4' is larger; '4' vs '6', '6' is larger.


# 4. Mixed Comparison: Let's compare strings with mixed characters (letters,
result_mixed = max('Banana', 'apple', 'Cherry', '123', '456')
print("The largest in the mixed set is:", result_mixed)
# '456' is the largest because:
# - 'B' (uppercase) is smaller than 'a' (lowercase),
# - 'C' (uppercase) is smaller than 'a' (lowercase),
# - Numbers are compared as characters, '4' is larger than 'a'.


# 5. Digging Deeper: If the first letters are the same, Python moves to the
result_deeper = max('cat', 'car')
print("The largest between 'cat' and 'car' is:", result_deeper)
# 'cat' is larger because after comparing the first two letters (which are t
# the third letters are 't' and 'r', and 't' comes after 'r'.
```

```
The largest alphabetically is: cherry
The largest considering case sensitivity is: apple
The largest number as a string is: 67
The largest in the mixed set is: apple
The largest between 'cat' and 'car' is: cat
```

- May call maximum with mixed types, such as `int`s and `float`s.

In [24]:
```
maximum(13.5, -3, 7)
```

Out[24]:
```
13.5
```

Function maximum's Definition

- Specify multiple parameters in a comma-separated list.
- To determine the largest value:
  - Assume that `value1` contains the largest value.
  - The first `if` statement then tests `value2 > max_value`, and if this condition is `True` assigns `value2` to `max_value`.
  - The second `if` statement then tests `value3 > max_value`, and if this condition is `True` assigns `value3` to `max_value`.
- Now, `max_value` contains the largest value.

Python's Built-In max and min Functions

- For many common tasks, the capabilities you need already exist in Python.

- Built-in `max` and `min` functions know how to determine the largest and smallest of their two or more arguments:

In [25]:
```
max('yellow', 'red', 'orange', 'blue', 'green')
```

Out[25]:
```
'yellow'
```

The next cell runs `min(15, 9, 27, 14)`.

In [26]:
```
min(15, 9, 27, 14)
```

Out[26]:
```
9
```

- Each also can receive an iterable argument, such as a list or a string.
- Using built-in functions or functions from the Python Standard Library's modules rather than writing your own can reduce development time and increase program reliability, portability and performance.
- Python's built-in functions and modules

Exercise (bowling)

A bowling night has three games. Write `series_average(g1, g2, g3)` that returns the mean of the three scores. Print the average of `140`, `160`, `180`.

In [ ]:
```
def series_average(g1, g2, g3):
    # mean of the three games
    return None

print(series_average(140, 160, 180))
```

Solution

In [1]:
```
def series_average(g1, g2, g3):
    return (g1 + g2 + g3) / 3

print(series_average(140, 160, 180))
```

```
160.0
```

4.9 Default Parameter Values

- You can specify that a parameter has a **default parameter value**.
- When calling the function, if you omit the argument for a parameter with a default parameter value, the default value for that parameter is automatically passed.

In [28]:
```
def rectangle_area(length=2, width=3):
    """Return a rectangle's area."""
    return length * width
```

- Specify a default parameter value by following a parameter's name with an `=` and a value.
- Any parameters with default parameter values must appear in the parameter list to the *right* of parameters that do not have defaults.

In [29]:
```
rectangle_area()
```

Out[29]:
```
6
```

The next cell runs `rectangle_area(10)`.

In [30]:
```
rectangle_area(10)
```

Out[30]:
```
30
```

The next cell runs `rectangle_area(10, 5)`.

In [31]:
```
rectangle_area(10, 5)
```

Out[31]:
```
50
```

The next cell runs
`##########################################################################`

In [32]:
```
##########################################################################
# (C) Copyright 2019 by Deitel & Associates, Inc. and                    #
# Pearson Education, Inc. All Rights Reserved.                           #
#                                                                        #
# DISCLAIMER: The authors and publisher of this book have used their     #
# best efforts in preparing the book. These efforts include the          #
# development, research, and testing of the theories and programs        #
# to determine their effectiveness. The authors and publisher make       #
# no warranty of any kind, expressed or implied, with regard to these    #
# programs or to the documentation contained in these books. The authors #
# and publisher shall not be liable in any event for incidental or       #
# consequential damages in connection with, or arising out of, the       #
# furnishing, performance, or use of these programs.                     #
##########################################################################
```

Exercise (taxi)

A taxi charges $3 plus $2 per mile. At night it adds $2. Write `taxi_fare(miles, night=False)` with that default. Return the fare. Print `taxi_fare(4)` and `taxi_fare(4, True)`.

In [ ]:
```
def taxi_fare(miles, night=False):
    # base + per-mile, then night extra when night is True
    return None

print(taxi_fare(4))
print(taxi_fare(4, True))
```

Solution

In [1]:
```
def taxi_fare(miles, night=False):
    fare = 3 + 2 * miles
    if night:
        fare = fare + 2
    return fare

print(taxi_fare(4))
print(taxi_fare(4, True))
```

```
11
13
```

4.10 Keyword Arguments

- When calling functions, you can use **keyword arguments** to pass arguments in `any` order.

In [33]:
```
def rectangle_area(length, width):
    """Return a rectangle's area."""
    return length * width
```

- Each keyword *argument in a call* has the form *parametername=value*.
- Order of keyword arguments does not matter.

In [34]:
```
rectangle_area(width=5, length=10)
```

Out[34]:
```
50
```

The next cell runs

```
##############################################################################
```

In [35]:
```
##############################################################################
# (C) Copyright 2019 by Deitel & Associates, Inc. and                        #
# Pearson Education, Inc. All Rights Reserved.                               #
#                                                                            #
# DISCLAIMER: The authors and publisher of this book have used their         #
# best efforts in preparing the book. These efforts include the              #
# development, research, and testing of the theories and programs            #
# to determine their effectiveness. The authors and publisher make           #
```

```
# no warranty of any kind, expressed or implied, with regard to these    #
# programs or to the documentation contained in these books. The authors #
# and publisher shall not be liable in any event for incidental or       #
# consequential damages in connection with, or arising out of, the       #
# furnishing, performance, or use of these programs.                     #
##########################################################################
```

Exercise (keyword call)

A hotel quotes `$120` a night. Weekend stays add `$40` total. Breakfast is `$15` per night. Write `room_rate(nights, weekend=False, breakfast=False)` and return the bill. Call it twice with **keywords** (do not pass a bare `True` in the second slot): `room_rate(nights=2, breakfast=True)` and `room_rate(nights=1, weekend=True, breakfast=True)`.

In [ ]:
```
def room_rate(nights, weekend=False, breakfast=False):
    # nights, optional weekend surcharge, optional breakfast
    return None

print(room_rate(nights=2, breakfast=True))
print(room_rate(nights=1, weekend=True, breakfast=True))
```

Solution (keyword call). Base `120 * nights`, then the two flags. Keyword names keep `weekend` and `breakfast` from swapping.

In [1]:
```
def room_rate(nights, weekend=False, breakfast=False):
    total = 120 * nights
    if weekend:
        total += 40
    if breakfast:
        total += 15 * nights
    return total

print(room_rate(nights=2, breakfast=True))
print(room_rate(nights=1, weekend=True, breakfast=True))
```

```
270
175
```

4.13 Scope Rules

- Each identifier has a `scope` that determines where you can use it in your program.
- Complete scope details.

Local Scope

- A local variable's identifier has **local scope**.

Global Scope

- Identifiers defined outside any function (or class) have **global scope**—these may include functions, variables and classes.

Accessing a Global Variable from a Function

In [36]:
```
x = 7
```

The next cell runs `def access_global():`.

In [37]:
```
def access_global():
    print('x printed from access_global:', x)
```

The next cell runs `access_global()`.

In [38]:
```
access_global()
```

```
x printed from access_global: 7
```

- By default, you cannot *modify* a global variable in a function
- Python creates a **new local variable** when you first assign a value to a variable in a function's block.
- In function `try_to_modify_global`'s block, the local `x` **shadows** the global `x`, making it inaccessible in the scope of the function's block.

In [39]:
```
x = 7
```

The next cell runs `def try_to_modify_global(x):`.

In [40]:
```
def try_to_modify_global(x):
    x = 3.5
    print('x printed from try_to_modify_global:', x)
    return x
```

The next cell runs `x = try_to_modify_global(x)`.

In [41]:
```
x = try_to_modify_global(x)
```

```
x printed from try_to_modify_global: 3.5
```

The next cell runs `x`.

In [42]:
```
x
```

Out[42]:
```
3.5
```

- To modify a global variable in a function's block, you must use a **`global`** statement to declare that the variable is defined in the global scope:

In [43]:
```
def modify_global():
    global x;
    x = 'hello'
    print('x printed from modify_global:', x)
```

The next cell runs `modify_global()`.

In [44]:
```
modify_global()
```

```
x printed from modify_global: hello
```

The next cell runs `x`.

In [45]:
```
x
```

Out[45]:
```
'hello'
```

Blocks vs. Suites

- When you create a variable in a block, it's *local* to that block.
- When you create a variable in a control statement's suite, the variable's scope depends on where the control statement is defined:
  - If it's in the global scope, any variables defined in the control statement have **global scope**.
  - If it's in a function's block, any variables defined in the control statement have **local scope**.

Shadowing Functions

- In the preceding chapters, when summing values, we stored the sum in a variable named `total`.
- If you define a variable named `sum`, it *shadows* the built-in function `sum`, making it inaccessible in your code.

In [46]:
```
sum = 10 + 5
```

The next cell runs `sum`.

In [47]:
```
sum
```

Out[47]:
```
15
```

The next cell runs `sum([10, 5])`.

In [48]:
```
sum([10, 5])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[48], line 1
----> 1 sum([10, 5])

TypeError: 'int' object is not callable
```

Statements at Global Scope

- Script statements at global scope execute as soon as they're encountered by the interpreter, whereas statements in a block execute only when the function is called.

Exercise (checkout)

A store keeps `tax_rate = 0.06` at **global** scope. Write `with_tax(price)` that uses that global rate (do not pass the rate in) and returns the price after tax. Print `with_tax(10)`.

In [ ]:
```
tax_rate = 0.06

def with_tax(price):
    # use the global tax_rate
    return None

print(with_tax(10))
```

Solution

In [1]:
```
tax_rate = 0.06

def with_tax(price):
    return price * (1 + tax_rate)

print(with_tax(10))
```

```
10.600000000000001
```

Comprehensive exercises

Exercise (comp-1)

Museum tickets: under 12 is free, members pay $10, everyone else $15. Write `museum_ticket(age, member=False)` with that default. Return the price. Call it as

`museum_ticket(10)`, `museum_ticket(30)`, and `museum_ticket(30, member=True)` (keyword).

In [ ]:
```
def museum_ticket(age, member=False):
    # free under 12; member price; otherwise full price
    return None

print(museum_ticket(10))
print(museum_ticket(30))
print(museum_ticket(30, member=True))
```

Solution

In [1]:
```
def museum_ticket(age, member=False):
    if age < 12:
        return 0
    if member:
        return 10
    return 15

print(museum_ticket(10))
print(museum_ticket(30))
print(museum_ticket(30, member=True))
```

```
0
15
10
```

Exercise (comp-2)

Keep a global `cap = 15` (maximum dollars off). Write `price_with_coupon(price, percent=0.1)` that takes `percent` off, but never subtracts more than `cap`. Use the global; do not pass `cap` in. Print `price_with_coupon(50)` and `price_with_coupon(400)`.

In [ ]:
```
cap = 15

def price_with_coupon(price, percent=0.1):
    # percent off, but not more than cap
    return None

print(price_with_coupon(50))
print(price_with_coupon(400))
```

Solution (comp-2). Discount is `price * percent`, then `min` with `cap`, then subtract. Default `percent` plus the global cap.

In [1]:
```
cap = 15

def price_with_coupon(price, percent=0.1):
```

```
    off = price * percent
    if off > cap:
        off = cap
    return price - off

print(price_with_coupon(50))
print(price_with_coupon(400))
```

```
45.0
385
```

Takeaways

- `def` names a function. A `return` sends a value back to the caller.
- Parameters are local. A default value is used only when the caller omits that argument.
- Keyword arguments let you pass values by name. A local name does not change a global of the same name unless you say `global`.
- Next class: **Modules**.