Functions

A grouping of predefined statements for repeated operations.

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

A ***function definition*** consists of the function's name and a block of statements.

A ***function call*** is an invocation of the function's name, causing the function's statements to execute.



A function may return one value using a ***return statement***.



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



a = int(input())



def compute\_square(a):



b = a \*\* 2



return b



print(f'{a} squared is {compute\_square(a)}')



* 7 squared is 49



A function can return only one item, not two or more (though a list or a tuple with multiple elements could be returned).

**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.

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.

Call a function named calc\_calories() passing the value 21 as an argument.

calc\_calories(21)



An argument is an expression like 21, num\_calories, num\_calories + 20, etc.



A parameter is like a variable definition. It cannot be an expression.



userNum + 5

print(get\_birthday\_age(42), get\_birthday\_age(20))

a comma creates a space between the two values, so the entire output is 43 21.

A function may have multiple parameters separated by commas.

A function definition with no parameters must still have the parentheses, as in: def calc\_something():

(99, 44+5) is legit argument.

### Hierarchical function calls

A function's statements may include function calls, known as ***hierarchical function calls*** or ***nested function calls***.

### Printing from a function

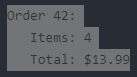
A function with no return statement is called a ***void function***, and such a function returns the value None.

old = int(input())

items = int(input())

price = float(input())

def print\_summary(old, items, price):

 print(f'Order {old}:')

print(f' Items: {items}')

print(f' Total: ${price:.2f}')

print\_summary(old, items, price)

### Dynamic and static typing

def add(x, y):

return x + y

1. add() function using two integer arguments, as in add(5, 7), which returns a value of 12.
2. add('Tora', 'Bora'), would concatenate the two strings and return 'ToraBora'

The function's behavior of adding together different types is a concept called ***polymorphism***.

( 5 \* 5)=25

( ‘x’ \* 5) = xxxxx

dynamic typing determines type of object in Python

C, C++, Java use static typing where each object must be defined

Polymorphism refers to how an operation depends on the involved object types. The behavior of an operator, such as + or \*, depends on the type of the operands.

***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 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.

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

Redundancy refers to writing the same code in multiple places. A programmer may want to write that code in a function, then call the function from multiple places in a program.

### Calling functions in expressions

y = square\_root(square\_root(16.0)) The inner square\_root(16.0) evaluates to 4.0. Then, the outer

square\_root(4.0) evaluates to 2.0, which y is then assigned with.

# Function stubs

To assist with the incremental development process, programmers commonly introduce ***function stubs***, which are function definitions whose statements haven't been written yet.

One approach is to use the ***pass*** keyword, which performs no operation except to act as a placeholder for a required statement.

***Scope*** is the area of code where a name is visible.

1. Built-in Scope - built-in names of Python, int( ), str( ), list( )
2. Global scope - contains all globally defined names outside of any function
3. local scope - within currently executing function

***Scope resolution*** The process of searching for a name in the available namespaces is called

Function Arguments

Arguments to functions are passed by object reference, a concept known in Python as ***pass-by-assignment***.



|  |  |  |  |
| --- | --- | --- | --- |
|  | Global namespace | Object | Local Namespace |
| def birthday(age):  '''Celebrate birthday!'''  age = age + 1 | birthday | 1. <funct> |  |
| timmy\_age = 7 | timmy\_age | 7 |  |
| birthday(timmy\_age)  print(f'Timmy is {timmy\_age}') |  |  |  |

* Timmy is 7 #8 doesn’t get passed through



def modify(num\_list):

num\_list[1] = 99



my\_list = [10, 20, 30]



modify(my\_list)



print(my\_list) # => [10, 99, 30]

def modify(num\_list):

num\_list[1] = 99 # Modifying only the copy

my\_list = [10, 20, 30]

modify(my\_list[:]) # Pass a copy of the list

print(my\_list) # my\_list does not contain 99!

* [10, 20, 30]

copied\_list = original\_list[:] or use dict.copy to create shallow copy

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other.

**Converting elements of a list to string or integer**

sample\_list\_int = [int(n) for n in sample\_list]

input = 2,3,4

sample\_list = input().split(",")

a = type(sample\_list[0])

print(sample\_list[0])

print(a)

* 2
* <class 'str'>

Trying to convert input( ) into a string upfront

input =2,3,4

* sample\_list = (int(input().split(",")))
* a = type(sample\_list[0])
* print(sample\_list[0])
* print(a)
* TypeError: int() argument must be a string, a bytes-like object or a real number, not 'list

sample\_list = input().split(",")

n = 10

for i in range(n):

*# i refers to new element*

    sample\_list = sample\_list+[i]

  sample\_list\_int = [int(n) for n in sample\_list]

print(sample\_list\_int)

* [2, 3, 4, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

sample\_list = input().split(",")

n = 10

for i in range(n):

  sample\_list = sample\_list+[i]

print(sample\_list)

* ['2', '3', '4', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

***Keyword arguments*** 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.

*Good practice is to use keyword arguments for any function containing more than approximately four arguments.*

*A common error is to place keyword arguments before all position arguments, which generates an exception.*

**Mixing keyword and default arguments**.

def split\_check(amount=10, num\_people=2, tax\_percentage=0.095, tip\_percentage=0.18):

these are default values

def number\_of\_pennies(dollars, pennies = 0):

    return dollars\*100 + pennies

print(number\_of\_pennies(int(input()), int(input()))) *# Both dollars and pennies*

print(number\_of\_pennies(int(input())))               *# Dollars only*

### Arbitrary arguments

**\*args**

A function definition can include an ***\*args*** parameter that collects optional positional parameters into an ***arbitrary argument list*** tuple.

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

**\*\*Kwags** 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.

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.

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.

def print\_stats(name, \*\*info):

    print(name, 'is:')

    for key, value in info.items():

        print(f'{key}: {value}')

print\_stats('John', age=10, gender='m')

* **John is:**
* **age: 10**
* **gender: m**

**Multiple Function Outputs**