CH-4: FUNCTIONS

September 2024

Introduction

Functions help in writing efficient, reusable, and organized programs.

Features of Functions

- **Divide and Conquer:** Break large programs into smaller, manageable parts.
- **Reusability:** Build new programs using existing functions as building blocks.
- **Function Calls:** Run code from multiple places without repetition.
- **Easy Maintenance:** Update oncechanges apply everywhere the function is used.
- **Modularity:** Makes code easier to read, test, and debug.

Defining Functions

- Youve called many built-in functions (`int`, `float`, `print`, `input`, `type`, `sum`, `len`, `min` and `max`) and a few functions from the statistics module (`mean`, `median and mode`).
- Each performed a single, well-defined task.
- Youll often define and call custom functions.

Defining a Custom Function

- A 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.
- The required parentheses contain the functions parameter lista comma-separated list of parameters representing the data that the function needs to perform its task.

- If the parentheses are empty, the function does not use parameters to perform its task.
- The indented lines after the colon (:) are the functions block.

Specifying a Custom Functions's Docstring

- The Style Guide for Python Code says that the first line in a functions block should be a docstring that briefly explains the functions purpose.

Returning a Result to a Functions Caller

- Functions are not run in a program until they are called or invoked in a program
- When a function finishes executing, it returns control to its callerthat is, the line of code that called the function.
- Function calls also can be embedded in expressions.

Function Characteristics

- Has a name
- Has (formal) parameters (0 or more)
- Has a docstring (optional but recommended)
- Has a body, a set of instructions to execute when function is called
- Returns something
  - Keyword `return`

What happens when you call a function

Function Call → Control passes to the function → Function executes its code → Returns a value (if any) → Control goes back to the caller.

HOW TO WRITE A FUNCTION

Function Name and Parameters:

```
def is_even(i):
```

Docstring:

```
""" Input:  i, a positive int.  Returns True if
i is even, otherwise False.  """
```

Function Body with Return Statements:

```
if i % 2 == 0:
    return True
else:
    return False
```

- There are two other ways to return control from a function to its caller:
  1. Using `return` without an expression:

```
def greet():
    print("Hello!")
    return
result = greet()
print("Returned value:", result)
------------OUTPUT--------------
Hello!
Returned value: None
```

  2. No return statement at all

```
def add(a, b):
    print("Sum is:", a + b)
    # no return statement      implicitly returns None
result = add(5, 3)
print("Returned value:", result)
----------OUTPUT-----------------------------------
Sum is: 8
Returned value: None
```

Modules, Packages, and Libraries (Optional)

Module:

- A module is a single file (with a `.py` extension) containing Python code.
- It can define functions, classes, variables, and can also include runnable code.
- Modules allow logical organization of Python code into separate files.
- Example: If you have a file named `math_utils.py`, it is a module that can be imported and used in another script.

```
# math_utils.py (a module)
def add(a, b):
    return a + b
```

- You can import and use the module like this:

```
# main.py (importing and using the module)
import math_utils
print(math_utils.add(3, 5))
```

Package:

- A package is a collection of modules organized in a directory with a special `__init__.py` file.
- This file is required to make Python treat the directory as a package, allowing the grouping of related modules together.
- A package can contain:
  - Multiple modules (Python files like `module1.py`, `module2.py`).
  - Sub-packages (directories containing more modules and an `__init__.py` file).
- Example: Suppose you have the following directory structure:

```
math_tools/              # This is a package
    __init__.py          # Makes it a package
    arithmetic.py        # Module 1
    algebra.py           # Module 2
```

- You can import modules from this package as follows:

```
from math_tools import arithmetic
```

Library:

- A library is a collection of modules and packages.
- It is a broader term that generally refers to a bundle of reusable code.
- A library could be made up of multiple modules and packages that provide various functions for a specific purpose.

- Standard Library: Python comes with a built-in standard library that includes many modules and packages (like math, os, sys, etc.) for common tasks.
- Third-Party Libraries: These are libraries created by other developers that you can install (usually through pip). For example, NumPy, Pandas, and Requests are all third-party libraries, and they often contain multiple modules and packages.

Functions with Multiple Parameters

- Define a `maximum` function that determines and return the largest of three values.

```
def maximum(v1, v2, v3):
    """Return the maximum of three values."""
    max_v = v1
    if v2 > max_v:
        max_v = v2
    if v3 > max_v:
        max_v = v3
    return max_v
-------SAMPLE OUTPUT--------------------
maximum(5,10,7)
10
```

- Python's Built-in `max` and `min` functions

```
max('red', 'green', 'blue', 'yellow', 'yellowX') #'yellowX'
min(15, 10, 20, 5, 6) # 5
```

For a list of Python's built-in functions and module, see Click here 👍

Random-Number Generation

- The random module, part of the Python Standard Library, provides a variety of tools for introducing randomness and chance in your programs.
- This module offers functions for generating random numbers, selecting random items from sequences, and working with randomness in different contexts, making it useful for simulations, games, data shuffling, and probabilistic algorithms.
- The randrange function generates an integer from the first argument value up to, but not including, the second argument value.
- Syntax: `random.randrange(start, stop, step)`
  - start(optional): The starting point of the range (inclusive). Defaults to 0 if not provided.
  - stop: The end of the range (exclusive). The generated number will be less than stop.
  - step(optional): The increment between numbers in the range. Defaults to 1.

```
import random
print(random.randrange(0, 20, 5))  # e.g., 10
```

Random-Number Generation

- Rolling a Six-Sided Die

```
import random
for roll in range(10):
    print(random.randrange(1,7), end = ' ')
---------------OUTPUT-----------------------
3 4 3 1 3 2 5 4 2 5 # First time
2 4 5 6 4 4 2 3 3 5 # Second time ...
```

- Tossing a coin

```
import random
for toss in range(10):
    t = random.randrange(0,2)
    if t == 0:
        print('H', end = ' ')
    else:
        print('T', end = ' ')
-----------OUTPUT------------
T T T T T T H H H H #First time
H H T T H H T H H T #Second time...
```

- Rolling a Six-Sided Die 6,000,000 Times

```
import random

# Number of rolls
num_rolls = 60_000_000

# Initialize individual counters
f1 = f2 = f3 = f4 = f5 = f6 = 0

# Roll the die num_rolls times
for _ in range(num_rolls):
    roll = random.randrange(1, 7)
    if roll == 1:
        f1 += 1
    elif roll == 2:
        f2 += 1
    elif roll == 3:
        f3 += 1
    elif roll == 4:
        f4 += 1
    elif roll == 5:
        f5 += 1
    else:
        f6 += 1
```

```python
# Print aligned results
print(f"{'Face':<10}{'Freq.':>15}{'Relative Freq.':>25}")
print(f"{1:<10}{f1:>15,}{f1/num_rolls:>25.6f}")
print(f"{2:<10}{f2:>15,}{f2/num_rolls:>25.6f}")
print(f"{3:<10}{f3:>15,}{f3/num_rolls:>25.6f}")
print(f"{4:<10}{f4:>15,}{f4/num_rolls:>25.6f}")
print(f"{5:<10}{f5:>15,}{f5/num_rolls:>25.6f}")
print(f"{6:<10}{f6:>15,}{f6/num_rolls:>25.6f}")
```

```
----------------------OUTPUT-----------------------
Face                 Freq.            Relative Freq.
1                 9,999,855                  0.166664
2                 9,997,401                  0.166623
3                 9,997,845                  0.166631
4                 9,999,933                  0.166666
5                10,003,343                  0.166722
6                10,001,623                  0.166694
```

- Seeding the Random-Number Generator for Reproducibility: You can use the random modules seed function to seed the random-number generator yourselfthis forces randrange to begin calculating its pseudorandom number sequence from the seed you specify.

```
import random
random.seed(10)
for i in range(8):
    print(random.randrange(1,5), end= ' ')
--------------OUTPUT-----------------------
2 3 4 2 2 1 1 2 2 2
########################################
random.seed(10)
for i in range(8):
    print(random.randrange(1,5), end= ' ')
--------------OUTPUT-----------------------
2 3 4 2 2 1 1 2 2 2 <- same output
```

Case Study: A Game of Chance

Simulate the popular dice game known as craps. Here is the requirements statement:

You roll two six-sided dice, each with faces containing one, two, three, four, five and six spots, respectively. When the dice come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first roll, you win. If the sum is 2, 3 or 12 on the first roll (called craps), you lose (i.e., the house wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, that sum becomes your point. To win, you must continue rolling the dice until you make your point (i.e., roll that same point value). You lose by rolling a 7 before making your point.

Simulating the Dice Game: Craps

- Roll two dice and calculate their sum.
- **First roll:**
  - Win if sum $= 7$ or 11
  - Lose if sum $= 2$, 3, or 12 ("craps")
  - Otherwise, the sum becomes your **point**.
- **Next rolls:**
  - Keep rolling until you match your point (win).
  - Lose if you roll a 7 before making your point.

```
import random

def roll_dice():
    die1 = random.randrange(1, 7)
    die2 = random.randrange(1, 7)
    return die1 + die2

sum_dice = roll_dice()
print("You rolled:", sum_dice)
```

```
if sum_dice in [7, 11]:
    print("You win!")
elif sum_dice in [2, 3, 12]:
    print("Craps! You lose.")
else:
    point = sum_dice
    print("Your point is:", point)
    while True:
        sum_dice = roll_dice()
        print("You rolled:", sum_dice)
        if sum_dice == point:
            print("You win!")
            break
        elif sum_dice == 7:
            print("You lose!")
            break
```

----------Sample OUTPUT----------

```
You rolled: 5
Your point is: 5
You rolled: 11
You rolled: 9
You rolled: 7
You lose!
```

Math Module Functions

- The math module defines functions for performing various common mathematical calculations.
- Import statement: `import math`
- Some math module functions are summarized belowyou can view the complete list at
  https://docs.python.org/3/library/math.html

Common Mathematical Functions in Python

Function	Description	Example
`ceil(x)`	Rounds x to the smallest integer not less than x.	`ceil(19.2)` is 20.0
`floor(x)`	Rounds x to the largest integer not greater than x.	`floor(9.2)` is 9.0
`sin(x)`	Trigonometric sine of x (in radians).	`sin(0.0)` is 0.0
`cos(x)`	Trigonometric cosine of x (in radians).	`cos(0.0)` is 1.0
`tan(x)`	Trigonometric tangent of x (in radians).	`tan(0.0)` is 0.0
`exp(x)`	Exponential function e^x.	`exp(1.0)` is 2.718282
`log(x)`	Natural logarithm of x (base e).	`log(2.718282)` is 1.0
`log10(x)`	Logarithm of x (base 10).	`log10(100.0)` is 2.0
`pow(x, y)`	x raised to power y (x^y).	`pow(2.0, 7.0)` is 128.0
`sqrt(x)`	Square root of x.	`sqrt(9.0)` is 3.0
`fabs(x)`	Absolute value of x always returns a float.	`fabs(-5.1)` is 5.1
`fmod(x, y)`	Remainder of x/y as a floating-point number.	`fmod(9.8, 4.0)` is 1.8

Default Parameter Values

- When defining a function, you can specify that a parameter has a default parameter value.
- When calling a function, if you **omit** the argument for a parameter with a **default parameter value**, the default value for that parameter is **automatically passed**.
- Parameter with with default parameter value must appear to the right of parameters that do not have defaults.

```
def greet(name, message="Hello"):
    """Demonstrates default parameter behavior"""
    print(message, name)
# Case 1: Using default value
greet("Alice")
# Case 2: Providing both arguments
greet("Bob", "Good evening")

# def wrong_example(greeting="Hi", name):
#     print(greeting, name)  #  Causes SyntaxError
------------OUTPUT--------------------------------
Hello Alice
Good evening Bob
```

Keyword Arguments

- When defining a function, you can use keyword arguments to pass arguments in *any order*.
- Each keyword *argument in a call* has the form `parametername = value`.
- In each function call, you must place keyword arguments after a functions positional argumentsthat is, any arguments for which you do not specify the parameter name.
- Positional arguments are assigned to the functions parameters left-to-right , based on the arguments positions in the argument list.
- Keyword arguments are also helpful for improving the readability of function calls, especially for functions with many arguments.

```
def greet(f_name, l_name, message):
    print(f"{message}, {f_name} {l_name}!")

greet(f_name="John", l_name="Doe", message="Welcome")
----------OUTPUT----------------------------------
Welcome, John Doe!
```

Arbitrary Argument Lists

- Function with arbitrary argument lists, such as `min` and `max` can receive *any* number of arguments.
- The functions documentation states that min has two required parameters (named `arg1` and `arg2`) and an optional third parameter of the form `*args`, indicating that the function can receive any number of additional arguments.
- The * before the parameter name tells Python to pack any remaining arguments into a tuple thats passed to the args parameter.

```
def my_min(arg1, arg2, *args):
    """Finds the smallest value among all given arguments."""
    smallest = arg1 if arg1 < arg2 else arg2
    # args holds any extra arguments beyond the first two
    for value in args:
        if value < smallest:
            smallest = value
    return smallest

print(my_min(5, 9))                    # Only two arguments
print(my_min(8, 3, 6, 2, 10, 4))       # Many arguments
print(my_min(12, 15, 7))               # Three arguments
```

Example-Arbitrary Argument List

Consider the following function that can receive any number of arguments:

```
def average(*args):
    return sum(args)/len(args)
```

- The parameter name args is used by convention, but you may use any identifier.
- If the function has multiple parameters, the *args parameter must be the rightmost parameter.
- In `average`, if `args` is empty, a `ZeroDivisionError` occurs.
- The *operator, when applied to an iterable argument in a function call, unpacks its elements.

Example-Arbitrary Argument List

Consider the following function:

```
def product(*a):
    pr = 1
    for x in a:
        pr*=x
    return pr
```

- The `*a` allows the function `product` to accept any number of positional arguments.
- When the function is called, all the arguments passed to `product` are collected into a tuple `a`.
- Inside the function `product`, the variable `a` is itself a **tuple** that contains all the arguments passed.

Methods: Functions That Belong to Objects

- A method is simply a function that you call on an object using the form `object_name.method_name(arguments)`

```
In [1]: s = 'Hello'
In [2]: s.lower()  # call lower method on string object s
Out[2]: 'hello'
In [3]: s.upper() # call upper method on string object s
Out[3]: 'HELLO'
In [4]: s
Out[4]: 'Hello'
```

- In the Object-Oriented Programming chapter, youll create custom types called classes and define custom methods that you can call on objects of those classes.

Scope Rules

- Each identifier has a scope that determines where you can use it in your program.
- For that portion of the program, the identifier is said to be in scope.

Local Scope

- A local variables identifier has local scope. Its in scope only from its definition to the end of the functions block.
- It goes out of scope when the function returns to its caller. So, a local variable can be used only inside the function that defines it.

```
def greet():
    name = "Alice"       # local variable
    print("Hello,", name)

greet()
print(name)  # Error: name is not defined (out of scope)
```

Global Scope

- Identifiers defined outside any function (or class) have global scopethese may include functions, variables and classes.
- Variables with global scope are known as global variables.
- Identifiers with global scope can be used in a `.py` file or interactive session anywhere after theyre defined.

```
message = "Welcome!"      # global variable

def show_message():
    print(message)         # accessible inside function

show_message()
print(message)             # also accessible outside
```

- In a functions block, the local variable shadows the global variable, making it inaccessible in the scope of the functions block.
- To modify a global variable in a functions block, you must use a global statement to declare that the variable is defined in the global scope
  `global x`

```
x = 10    # global variable

def change():
    global x
    x = 20    # modifies the global x
    print("Inside function:", x)

change()
print("Outside function:", x)
------OUTPUT-----------------------
Inside function: 20
Outside function: 20
```

Blocks vs. Suites

- If the control statement is in the global scope, then any variables defined in the control statement have global scope.
- If the control statement is in a functions block, then any variables defined in the control statement have local scope.

- If you define a variable named `sum`, it shadows the built-in function, making it inaccessible in your code.
- When you execute the following assignment, Python binds the identifier sum to the int object containing 15.
- At this point, the identifier `sum` no longer references the built-in function.

```
In [1]: sum = 10 + 5
In [2]: sum
Out[2]: 15
```

- So, when you try to use sum as a function, a `TypeError` occurs:

```
In [3]: sum([10, 5])
```

- Statements at Global Scope: Script statements at global scope execute as soon as theyre encountered by the interpreter, whereas statements in a block execute only when the function is called.

import: A Deeper Look

- Youve imported modules (such as `math` and `random`) with a statement like:
  `import` *`module_name`*
  then accessed their features via each modules name and a dot (.).
- Also, youve imported a specific identifier from a module (such as the decimal modules Decimal type) with a statement like:
  `from module_name import` *`identifier`*
  then used that identifier without having to precede it with the module name and a dot (.).

Importing Multiple Identifiers from a Module

```
from modulename import name1, name2, ...
```

Caution: Avoid Wildcard Imports

- You can import all identifiers defined in a module with a wildcard import of the form

  ```
  from modulename import *
  ```

- This makes all of the modules identifiers available for use in your code.
- Importing a modules identifiers with a wildcard import can lead to subtle errorsits considered a dangerous practice that you should avoid.

Binding Names for Modules and Module Identifiers

```
import math as m

print(m.sqrt(25)) # 5.0
-------------------------------------
from math import pow as p

print(p(2,3)) # 8.0
```

Passing Arguments to Functions: A Deeper Look

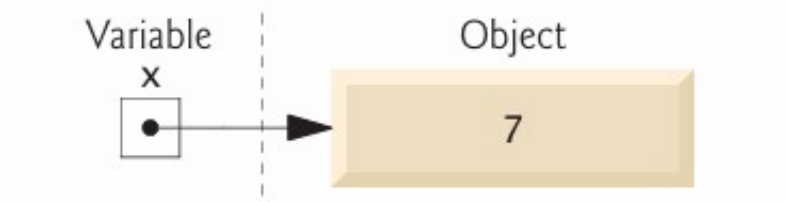
- In many programming languages, there are two ways to pass arguments-
  - pass by value (or call by value)
  - pass by reference (or call by reference)
- Pass-by-value: function gets a copy; changes don't affect the original.
- Pass-by-reference: function can modify the caller's value if mutable.
- Python arguments are always passed by reference.
- Also called pass-by-object reference, since everything is an object.

```
def modify_list(lst):
    lst.append(100)
    print("Inside function:", lst)

my_list = [1, 2, 3]
modify_list(my_list)
print("Outside function:", my_list)
------------OUTPUT-----------------
Inside function: [1, 2, 3, 100]
Outside function: [1, 2, 3, 100]
```

Memory Addresses, References and Pointers

- You interact with an object via a reference, which behind the scenes is that objects address (or location) in the computers memorysometimes called a pointer in other languages.
- After an assignment like
  `x = 7`
  the variable x does not actually contain the value 7. Rather, it contains a reference to an object containing 7 (and some other data) stored elsewhere in memory.
- You might say that x points to (that is, references) the object containing 7



Built-In Function id and Object Identities

- `In [1]: x = 7` x points to the integer object 7.
- Every object has a unique address in memory.
- Use `id(x)` to get a unique integer identifying the object:

```
In [2]: id(x)
Out[2]: 4350477840
```

- The `id` value is the objects identity; no two objects share the same identity.

Passing Objects to a Function

- **Object identity:** `is` checks if two variables refer to the same object.

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]
print(x is y)   # True  (same object)
print(x is z)   # False (different objects with same contents)
```

- **Immutable arguments:** Values like `int`, `float`, `str`, `tuple` cannot be modified inside functions.
- **Mutable arguments:** Objects like `list`, `dict` can be changed inside functions.

```
# Immutable example
def modify(x):
    x = x + 1
a = 10
modify(a)
print(a)    # 10 (unchanged)

# Mutable example
def add_item(lst):
    lst.append(4)
nums = [1, 2, 3]
add_item(nums)
print(nums)  # [1, 2, 3, 4]
```

Function-Call Stack

- The function-call stack is a **Last-In, First-Out (LIFO)** data structure used to manage active function calls.
- Each time a function is called, Python *pushes* a **stack frame** (or activation record) onto the stack.
- A **stack frame** stores:
  - functions parameters and local variables,
  - return address (where to go back after execution).
- When the function finishes, its frame is *popped*, and control returns to the callers frame.
- The frame at the **top** of the stack is always the function **currently executing**.

Visualization:

`main()` $\rightarrow$ calls $\rightarrow$ `A()` $\rightarrow$ calls $\rightarrow$ `B()`

Stack (top bottom): [B()] [A()] [main()]

- Each functions **local variables** and **parameters** live inside its own stack frame.
- These variables:
  - exist while the function executes,
  - remain safe even if other functions are called,
  - disappear once the function returns (frame popped).
- Recursive functions create a new stack frame for each recursive call, preserving the previous calls state.

```
def funcA():
    funcB()

def funcB():
    funcC()

def funcC():
    print("End of chain")

funcA()  # Stack grows: funcA -> funcB -> funcC
-------OUTPUT-------
End of chain
```

- Stack Overflow: occurs when too many nested calls (often from infinite recursion) exhaust memory.

Functional-Style Programming

- Python offers functional-style features that help you write code which is less likely to contain errors, more concise and easier to read, debug and modify.
- Functional style programs also can be easier to parallelize to get better performance on todays multicore processors.

Functional-style programming topics

avoiding side effects
closures
declarative programming (4)
decorators (10)
dictionary comprehensions (6)
`filter/map/reduce` (5)
`functools` module
generator expressions (5)
generator functions (12)
higher-order functions (5)
immutability (4)
internal iteration (4)
iterators (3)
`itertools` module (18)
`lambda` expressions (5)
lazy evaluation (5)
list comprehensions (5)
`operator` module (5, 13, 18)
pure functions (4)
`range` function (3, 4)
reductions (3, 5)
set comprehensions (6)

What vs. How

- Functional-style programming lets you simply say **what** you want to do. It hides many details of **how** to perform each task.
- Typically, **library code** handles the **how** to do.
- In `for` statement in many other programming languages, you must specify all the details of counter controlled iteration: a control variable, its initial value, how to increment it and a loop continuation condition that uses the control variable to to determine whether to continue iterating.
- This style of iteration is known as external iteration and is **error-prone**.
- External iteration **mutates** (i.e., modifies) the control variable, and the `for` statement's suit often mutates other variables as well.

- Functional-style programming emphasizes immutability. That is, it avoids operations that modify variables values.
- Youve already used list, string and built-in function **range** *iterators* with the `for` statement, and several *reductions* (functions `sum`, `len`, `min` and `max`).
- Specifying what, but not how, is an important aspect of internal iterationa key functional-style programming concept.
- Stating what you want done rather than programming how to do it is known as declarative programming.
- In pure functional programming language you focus on writing pure functions. A pure functions result depends only on the argument(s) you pass to it.
- Also, given a particular argument (or arguments), a pure function always produces the same result.
- Also, a pure function does not have side effects.

Introduction to Data Science: Measures of Dispersion

- A measure of dispersion refers to the statistical tools used to describe the spread or variability of data points in a dataset.
- It quantifies how much the data deviates from the central value (like the mean or median).

Common measures of dispersion include:

- Range: The difference between the highest and lowest values.
- Variance: The average of the squared differences from the mean.
  `statistics.pvariance()`
- Standard Deviation: The square root of variance, representing the average distance of data points from the mean.
  `statistics.pstdev()`

Q.1 (Modified average Function) The average function we defined earlier can receive any number of arguments. If you call it with no arguments, however, the function causes a ZeroDivisionError. Reimplement average to receive one required argument and the arbitrary argument list argument *args, and update its calculation accordingly. Test your function. The function will always require at least one argument, so youll no longer be able to get a ZeroDivisionError. When you call average with no arguments, Python should issue a TypeError indicating "average() missing 1 required positional argument."

Q.2 (Guess the Number) Write a script that plays guess the number. Choose the number to be guessed by selecting a random integer in the range 1 to 1000. Do not reveal this number to the user. Display the prompt "Guess my number between 1 and 1000 with the fewest guesses:". The player inputs a first guess. If the guess is incorrect, display "Too high. Try again." or "Too low. Try again." as appropriate to help the player zero in on the correct answer, then prompt the user for the next guess. When the user enters the correct answer, display "Congratulations. You guessed the number!", and allow the user to choose whether to play again.

Q.3 (Guess-the-Number Modification) Modify the previous exercise to count the number of guesses the player makes. If the number is 10 or fewer, display "Either you know the secret or you got lucky!" If the player makes more than 10 guesses, display "You should be able to do better!" Why should it take no more than 10 guesses? Well, with each good guess, the player should be able to eliminate half of the numbers, then half of the remaining numbers, and so on. Doing this 10 times narrows down the possibilities to a single number. This kind of halving appears in many computer science applications. For example, in the Computer Science Thinking: Recursion, Searching, Sorting and Big O chapter, well present the high-speed binary search and merge sort algorithms, and youll attempt the quicksort exerciseeach of these cleverly uses halving to achieve high performance.

Q.4 (Date and Time) Pythons datetime module contains a datetime type with a method today that returns the current date and time as a datetime object. Write a parameterless `date_and_time` function containing the following statement, then call that function to display the current date and time:
print(datetime.datetime.today())
On our system, the date and time display in the following format:
`20180608 13:04:19.214180`

Q.5 You have already simulated the popular dice game known as craps. whose requirements statement is:

You roll two six-sided dice, each with faces containing one, two, three, four, five and six spots, respectively. When the dice come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first roll, you win. If the sum is 2, 3 or 12 on the first roll (called craps), you lose (i.e., the house wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, that sum becomes your point. To win, you must continue rolling the dice until you make your point (i.e., roll that same point value). You lose by rolling a 7 before making your point.

Now, estimate the probability of wining this game.

Q.6 **(Simulation: The Tortoise and the Hare)** In this problem, youll re-create the classic race of the tortoise and the hare. Youll use random-number generation to develop a simulation of this memorable event.

Our contenders begin the race at square 1 of 70 squares. Each square represents a position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

A clock ticks once per second. With each tick of the clock, your application should adjust the position of the animals according to the rules in the table below. Use variables to keep track of the positions of the animals (i.e., position numbers are 170). Start each animal at position 1 (the starting gate). If an animal slips left before square 1, move it back to square 1.

Animal	Move type	Percentage of the time	Actual move
Tortoise	Fast plod	50%	3 squares to the right
	Slip	20%	6 squares to the left
	Slow plod	30%	1 square to the right
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares to the right
	Big slip	10%	12 squares to the left
	Small hop	30%	1 square to the right
	Small slip	20%	2 squares to the left

Create two functions that generate the percentages in the table for the tortoise and the hare, respectively, by producing a random integer i in the range $1 \leq i \leq 10$. In the function for the tortoise, perform a fast plod when $1 \leq i \leq 5$, a slip when $6 \leq i \leq 7$ or a slow plod when $8 \leq i \leq 10$. Use a similar technique in the function for the hare.

Begin the race by displaying **BANG !!!!!**

AND THEY'RE OFF !!!!!

Then, for each tick of the clock (i.e., each iteration of a loop), display a 70-position line showing the letter **"T"** in the position of the tortoise and the letter **"H"** in the position of the hare. Occasionally, the contenders will land on the same square. In this case, the tortoise bites the hare, and your application should display "OUCH!!!" at that position. All positions other than the "T", the "H" or the **"OUCH!!!"** (in case of a tie) should be blank.

After each line is displayed, test for whether either animal has reached or passed square 70. If so, display the winner and terminate the simulation.

If the tortoise wins, display **TORTOISE WINS!!! YAY!!!** If the hare wins, display Hare wins. **Yuch**. If both animals win on the same tick of the clock, you may want to favor the tortoise (the underdog), or you may want to display **"It's a tie"**. If neither animal wins, perform the loop again to simulate the next tick of the clock. When youre ready to run your application, assemble a group of fans to watch the race. Youll be amazed at how involved your audience gets!

Q.7 **(Arbitrary Argument List)** Calculate the product of a series of integers that are passed to the function product, which receives an arbitrary argument list. Test your function with several calls, each with a different number of arguments.

Q.8 **(Computer-Assisted Instruction)** Computer-assisted instruction (CAI) refers to the use of computers in education. Write a script to help an elementary school student learn multiplication. Create a function that randomly generates and returns a tuple of two positive one-digit integers. Use that functions result in your script to prompt the user with a question, such as

```
How much is 6 times 7?
```

For a correct answer, display the message "Very good!" and ask another multiplication question. For an incorrect answer, display the message "No. Please try again." and let the student try the same question repeatedly until the student finally gets it right.

Q.9 **(Computer-Assisted Instruction: Reducing Student Fatigue)** Varying the computers responses can help hold the students attention. Modify the previous exercise so that various comments are displayed for each answer. Possible responses to a correct answer should include 'Very good!', 'Nice work!' and 'Keep up the good work!' Possible responses to an incorrect answer should include 'No. Please try again.', 'Wrong. Try once more.' and 'No. Keep trying.' Choose a number from 1 to 3, then use that value to select one of the three appropriate responses to each correct or incorrect answer.

Q.10 **(Computer-Assisted Instruction: Difficulty Levels)** Modify the previous exercise to allow the user to enter a difficulty level. At a difficulty level of 1, the program should use only single-digit numbers in the problems and at a difficulty level of 2, numbers as large as two digits.

Q.11 **(Computer-Assisted Instruction: Varying the Types of Problems)** Modify the previous exercise to allow the user to pick a type of arithmetic problem to study1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only, 4 means division problems only (avoid dividing by 0) and 5 means a random mixture of all these types.