CS5001 Synthesis Assignment 2

Shang Xiao

April 8, 2022

Contents

1 For Loops **3**
- 1.1 For Loops . . . 3
- 1.2 Counter Controlled Loops . . . 4
- 1.3 Iterating Over a List . . . 5
- 1.4 Range Over a List . . . 7

2 Recursion **9**
- 2.1 Recursion . . . 9
- 2.2 Why Do We Use Recursion? . . . 9
- 2.3 Functions Calling Themselves . . . 11
- 2.4 Mechanics/Mathematics of Recursion . . . 12
- 2.5 What Recursion Is NOT Good For . . . 14
- 2.6 Debugging Recursion . . . 15

3 Exception Handling **16**
- 3.1 Error Handling . . . 16
- 3.2 Defensive Programming . . . 17
- 3.3 Asking Permission - Raising Errors . . . 17
- 3.4 Asking Permission - Repetition . . . 18
- 3.5 Asking Forgiveness . . . 19
- 3.6 Understanding Execution Paths . . . 21
- 3.7 Files . . . 22
- 3.8 Collecting Data . . . 23

4 Classes and Objects **25**
- 4.1 Classes and Objects . . . 25
- 4.2 Intro to Classes . . . 26
- 4.3 Working with Objects . . . 27
- 4.4 Defining Classes . . . 28
- 4.5 Constructors . . . 29
- 4.6 Printing Objects . . . 30
- 4.7 Testing Classes . . . 31

1 For Loops

1.1 For Loops

Explanation

Sometimes, we often need to iterate through all the list elements, performing the same operation on each one. For example, shifting each interface element to the same distance; for a list containing numbers, you may need to perform the same statistical operation on each component. Each title is a list of articles on a website that may need to be displayed. If you need to perform the same operation for each element in the list When you need to complete the same process on each part of the list, you can use a for loop in Python.

Suppose we have a list of students and we need to print out the name of each one of them. We can get each word in the list separately and get each word in the list separately, but this leads to several problems. For example, if the list is long, it will contain many duplicate codes. In addition, the code would have to be modified whenever the length of the list changes. By using a for loop, you can let Python handle these problems. The following loop is used to print all the names in the students' list.

```
students = ['alice', 'david', 'carolina']
for student in students:
    print(student)
```

This line of code tells Python to take a name from the list of students and store it in the variable students. Finally, Python prints the last name into the variable students (see). This way, for each word in the list, Python will repeat the lines of code at and. You can read the code this way: the term is printed out for each students in the list of students. The output is all the names in the list.

```
alice
david
carolina
```

Exercise

Why do we need a for loop when we already have a while loop in Python?

Solution

Generally speaking, for loop is more commonly used for beginners than while loop, but in fact, while loop has more functions than for loop and all for loops can be represented by while loops, but not all while loops can be represented by for loops. But the for loop is easy to write, express and explain. We can easily

get the result we want by using range functions and iterating without worrying if we forgot to add one to some parameter.

1.2 Counter Controlled Loops

Explanation

The Python function range() allows you to easily generate a series of numbers. For example, you can use a function like this range() to print a range of numbers.

```
for i in range(0,10):
    print(i,end=', ')
```

The above code seems like it should print the numbers 0 to 10, but it does not actually print the number 10:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

In this example, range() just prints the numbers 0 to 9, resulting from the differential behavior you often see in programming languages. The range() function tells Python to start counting from the first value you specify and stop when it reaches the second value you select, so the output does not contain the second value (in this case, 5). To print the numbers 0 to 10, you need to use the function range(0,11).

```
for i in range(0,11):
    print(i,end=', ')
```

Thus, the output will start at 0 and end at 10.

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

With range, we can specify how many loops to make.

Exercise

1. Write the result of the following program:

```
for i in range(10):
   print(i, end = ", ")
```

2. After running the following program, what is the value of the final i?

```
for i in range(10):
   print(i, end = ", ")
```

3. Read the input ten times and output the input in reverse order.

Solution

1.
```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

2. 9

3.
```
ay = []
for i in range(0,10):
    ay.append(input())
for i in range(9,-1,-1):
    print(ay[i],end=',')
```

1.3 Iterating Over a List

Explanation

A list consists of a series of elements arranged in a specific order. You can create lists containing all the letters of the alphabet, the numbers 0 to 9, or all the names of family members, or you can add anything to a list where the elements can have no relationship to each other. Given that lists usually contain multiple elements, we can use a for loop to iterate through them.

Even strings are iterable, and we can use loops to operate on characters within lines. For example:

```
for i in "NEU":
    print(i)
```

The result is that:

```
N
E
U
```

In a for loop, any operation can be performed on each element. Here's an extension of the previous example to count the sum of the string lengths for each student's name.

```
students = ['alice', 'david', 'carolina']
total = 0
for student in students:
    total += len(student)
print(total)
```

With the above code, we can live the sum of the length of all students' names.

For strings, we can also use split to group strings for manipulation. The split() method splits the string into multiple parts using spaces as separators

and stores all these parts in a list. The result is a list containing all the words in the string, although some words may contain punctuation.

Here is a example:

```
for w in "Now is the war of ...".split():
    print(w)
```

The result is:

```
Now
is
the
war
of
...
```

Of course, we can also specify how to separate the splits to achieve the desired effect.

```
for w in "Life is short, you need Python".split("o"):
    print(w)
```

As we would expect, the results are shown below:

```
Life is sh
rt, y
u need Pyth
n
```

Exercise

1. Write the result of the following cycle:

```
for w in "python is the best language in the world.".split("i"):
    print(w)
```

2. Given a string, arrange the string in reverse order and store the result in another variable.

Solution

1.

```
python
s the best language
n the world.
```

2.

```
s = "abcdefg"
```

```
ans = ""
for c in s:
    ans = c+ans
print(ans)
```

1.4 Range Over a List

Explanation

Lists are ordered sets, so to access any element of a list, tell Python the location or index of that element. To access a list element, point to the list's name, point to the element's index, and put it in square brackets. For example, the following code extracts the first bike from the list bicycles:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0])
```

In Python, the first list element has an index of 0, not 1. This is true in most programming languages. In addition we can get the length of the sequence by using the len function. So we can access the array by index via a for a loop using range.Here is a example:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
for i in range(len(bicycles)):
    print(bicycles[i])
```

The result is shown as follow:

```
trek
cannondale
redline
specialized
```

And in these cases, we can use while instead of for.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
i = 0
while i<range(len(bicycles)):
    print(bicycles[i])
    i+=1
```

Comparing the two codes, we find that using a for loop makes the code more concise and easy to understand. But at the same time, not all loops can be replaced by while loops in some cases.

Exercise

1. Combine two arranged arrays into one array in size order. And output them in order from largest to smallest.

2. Combine three sequences from small to large into an ordered sequence and maximize the efficiency of the algorithm.And output them in order from largest to smallest.

Solution

1.
```
a = [1,2,3,4]
b = [3,4,7,8,9]
ans = []
j = 0
for i in range(0,len(a)):
    while a[i] > b[j] and j<len(b):
        ans.append(b[j])
        j+=1
    ans.append(a[i])

for i in range(j,len(b)):
    ans.append(b[i])

for i in range(len(ans)-1,-1,-1):
    print(ans[i],end=", ")
```

2.
```
def combine(a,b):
    ans = []
    j = 0
    for i in range(0,len(a)):
        while a[i] > b[j] and j<len(b):
            ans.append(b[j])
            j+=1
        ans.append(a[i])

    for i in range(j,len(b)):
        ans.append(b[i])
    return ans

a = [1,5,6,78546,5436354]
b = [13,435135,3443534,532532453]
c = [1,2,3,4,5,6,7,7,8,9,9]
d = combine(a,b)
ans = combine(c,d)

for i in range(len(ans)-1,-1,-1):
    print(ans[i],end=', ')
```

2 Recursion

2.1 Recursion

Explanation

Recursion means decomposing the problem into countless smaller problems, solving them layer by layer, and finally reaching the critical point after solving the last problem that needs to call its own function to handle, there is back: the result of the solution is returned to the original point, and the original problem is solved.

The basic idea of recursion is to take a problem of large size and decompose it into multiple sub-problems of smaller size to solve, and each sub-problem can continue to be split into multiple smaller sub-problems.

The most important point is to assume that the subproblem has been solved, and now the current problem has to be solved based on the solved subproblem; or rather, the subproblem must be solved first, and then the current problem based on the subproblem Or it can be understood this way: recursion solves multiple problems that have dependency order relations. Let's assume that an abstract problem has two point-in-time elements: start processing and end processing. Then the order of recursive processing is that the problem that starts processing first ends processing last.

Assume the following problem dependencies:

$$[A]----depends \quad on----->[B]----depends \quad on----->[C] \quad (1)$$

Our ultimate goal is to solve problem A. Then the three problems are processed in the following order.

- Start processing problem A.
- Start processing problem B since A depends on B.
- Start processing problem C since B depends on C.
- conclude processing problem C.
- Ends processing of problem B.
- Ends processing of problem A‘

Exercise

Solution

2.2 Why Do We Use Recursion?

Explanation

Recursion as an algorithm is widely used in programming languages. A procedure or function has a way to call itself directly or indirectly in its definition

or description. It usually transforms a large and complex problem layer by layer into a smaller scale problem similar to the original problem to solve, and the recursive strategy requires only a small number of procedures to describe the multiple iterations required to solve the problem, greatly reducing the amount of code in the program. The power of recursion lies in the ability to define an infinite set of objects with a finite number of statements. In general, recursion requires a boundary condition, a recursive forward segment, and a recursive return segment. When the boundary condition is not satisfied, recursion advances; when the boundary condition is satisfied, recursion returns.

The recursive function call is shown in the figure:

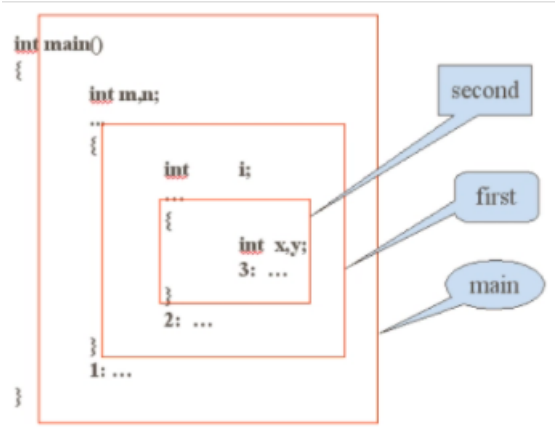

Figure 1: Recursion

Conditions required to constitute recursion.

- the subproblem must be the same thing as the original problem, and simpler.
- the call itself cannot be unlimited and must have an exit, reduced to a non-recursive condition to handle.

Exercise

Judge the following statements and explain why.

1. Recursion is the process of continuously solving subproblems.
2. Nature inspires the invention of recursion.

Solution

1. True. Recursion is a solution to a problem implemented by calling one of its functions a subroutine. We will be surprised at the beginning how

we can implement a function that calls itself. The beauty of it is that each time the recursive function calls itself, it reduces the given problem to a subproblem. The recursive call will continue until the subproblem is solved and there is no more recursion.

2. True. For example, the mirror looks into the mirror, and the inside is continuously recursively extended to infinity.

2.3 Functions Calling Themselves

Explanation

The basic idea of recursion is to solve problems of large size by transforming them into similar subproblems of small size. When the function is implemented, because the method to solve the large problem and the method to solve the small problem are often the same method, a situation arises where the function calls itself. Also this problem-solving function must have an obvious end condition so that infinite recursion does not arise.

Not all problems can be solved by recursion. There are two conditions that must be met in order to use recursion.

The idea of recursion is that in order to solve the current problem F(n), one needs to solve the problem F(n-1), and the solution of F(n-1) depends on the solution of F(n-2) ..... is thus decomposed layer by layer, into When the smallest events are solved, the higher-level events can be solved. This "layer-by-layer decomposition, layer-by-layer merging" approach constitutes the idea of recursion.

The main thing to do with recursion is to find the exit of recursion and the way of recursion. So recursion is usually divided into two parts: the recursive way and the recursive termination condition (the solution of the minimum event). These two parts are the key to recursion!

The way of recursion is the recursive formula, i.e., the decomposition of the problem, and also the rules for convergence to the recursive termination condition. And the recursive termination condition is usually the resulting solution of the minimum event. The purpose of the recursive termination condition is to prevent the recursion from going on indefinitely, and eventually it must "stop".

In Python, recursion is achieved by calling itself in the function

Exercise

1. Outputs the reverse order of a string using recursion.

2. Recursively calculate the sum of 1 to 100 and return the result.

Solution

1.
```
s = "https://northeastern.instructure.com/"

```

```
3
4  def reverse(s, i):
5      if i >= len(s):
6          return
7      reverse(s,i+1)
8      print(s[i])
9
10
11 reverse(s, 0)
```

2.

```
1  def sum(a):
2      if a == 100:
3          return a
4      ans = sum(a+1)+a
5      return ans
6
7  print(sum(1))
```

2.4 Mechanics/Mathematics of Recursion

Explanation

A recursive formula is when the recursive formula contains only the terms in the series and no constant terms or other terms. The formulaic way of recursive programming is a simple and effective design idea that focuses the difficulties of programming and program understanding on recursive formulas. Programs designed from recursive procedures have a standard branching structure and are much simpler to write and understand. In this section, we master the implementation of two recursive formulas, the factorial, and the Fibonacci series, respectively.Both are defined as follows:

- Factorial: The factorial of a positive integer is the product of all positive integers less than and equal to that number, and the factorial of 0 is 1. The factorial of a natural number n is written n!
- Fibonacci sequence (Fibonacci sequence), also known as the golden mean sequence. It was introduced by mathematician Leonardoda Fibonacci as an example of rabbit reproduction, so it is also called the "rabbit sequence", referring to a sequence of numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34,... ...In mathematics, the Fibonacci sequence is defined recursively as follows: $F(1) = 1, \quad F(2) = 1, F(n) = F(n-1) + F(n-2) \quad (n >= 2, \quad n \in N^*)$.

We can call the function itself to achieve the cumulative multiplication of factorials and the accumulation of Fibolacci. After listening to the instructor and combining my own knowledge base, I implemented the two codes as follows:

- ```
  def factorial(n):
      if n==1:
          return 1
      else:
          return n*factorial(n-1)
  ```

- ```
  def fab(n):
      n1 = 1
      n2 = 1
      n3 = 1

      if n < 1:
          print('wrong input')
          return -1
      while (n-2) > 0:
          n3 = n2 + n1
          n1 = n2       #Calculate the next iteration by shifting n1
               and n2 back in sequence, n2 to the current n1, and the
               previous n3 to n2, repeating the operation to sum
          n2 = n3
          n -=1         #Calculate and reduce n once until n is 2,
               then exit the loop
      return n3
  ```

Exercise

1. Recursively computes the sum of the first n positive integers.
2. Recursive computing $1! + 2! + 3! + ... + n!$.

Solution

1. ```
   def sum(n):
       if n == 0:
           return 0
       return n+sum(n-1)

   sum(10)
   ```

2. ```
   def sum(n):
       if n == 1:
           return [1,1]
       a = sum(n-1)
       return [a[0]*n,a[1]+a[0]*n]

   ```

```
print(sum(3)[1])
```

2.5 What Recursion Is NOT Good For

Explanation

Recursion requires space on the stack, and the size of the stack is fixed, which means that it cannot be recursive indefinitely. At some point in the recursion, there will be no more space at the top of the stack.A full stack is like a cupboard being crammed full of space and not being able to add a plate. And recursion is a function call itself, and function calls consume time and space. Each function call requires space on the memory stack to hold parameters, return values, and temporary variables, and pressing and popping data onto the stack takes time, so it reduces efficiency.

Recursion requires making many function calls, each of which requires setting up a stack frame and passing arguments, all of which add time overhead that is not present in the loop. In most cases, these overheads are not significant in modern computers. In most cases, these overhead effects are not significant in modern computers. But if your code executes frequently, say millions or even hundreds of millions of times in a short period, you must be concerned about function call performance. Recursion is more potent than loops in that recursive functions maintain a stack that holds the current state of each recursive call, allowing the process to obtain the results of subproblems and then continue processing.

The essence of recursion is to decompose a problem into two or more minor issues, where there are overlapping parts, i.e., repeated calculations, such as the recursive implementation of the Fibonacci sequence.Repeated calls increase time expenses, but of course we can reduce such time and space expenses by pruning and memorizing searches.

Exercise

Judge the following statements.

1. Recursion is a good choice for effective implementation of mathematical definitions.

2. Recursive solutions can cause the program to run out of memory if the recursion is too large.

Solution

1. False. Stack memory is used for recursion. Since the stack maintains information about each function call until it is released after the function returns, this takes up a considerable amount of space, mainly if many recursive calls are used in the program. On top of that, it takes some

time to generate and destroy stack frames because of the large amount of information that needs to be saved and restored.

2. True. Stack memory is used for recursion and the size of stack is small.

2.6 Debugging Recursion

Explanation

Debugging recursive functions is more complicated than general functions. Here, I summarize some debugging techniques.

- use counter breakpoints. Most IDEs provide a counter type of breakpoint; that is, the breakpoint will be triggered only when the number of triggers reaches the value set by the counter. You can set this type of breakpoint to assist debugging for recursive functions such as retry type, which can be expected to execute the number of times about the corresponding business logic. 2.

- Use conditional breakpoints. As the name suggests, a conditional breakpoint is a breakpoint that will be triggered only when the set condition is met. We can set the breakpoint starting conditions flexibly with this type of breakpoint. For example, we can put a breakpoint to ensure that an intermediate value meets a specific situation.

- debugging key points. Unit testing of recursive functions should focus on the following key points.

  (a) end conditions. Especially for recursive functions with multiple end conditions, debugging should pay attention to all end conditions and their combinations.

  (b) intermediate values. The intermediate values passed as arguments should be kept reasonably, as deviations can lead to unexpected results in later recursive calls. Therefore, debugging should pay attention to the changes in the intermediate values passed during the recursive calls.

Exercise

Find the problem of the following code which calculates the sum of positive integers less than n using debugging.

```
1 def sum(n):
2     if n == 1:
3         return 0
4     return n+sum(n-1)
5
6 sum(10)
```

Solution

I add *print* to see the values in the intermediate process. So I find that the function returns an error value when recursing to the lowest level. My debugging code is here:

```
def sum(n):
    if n == 1:
        return 0
    t = sum(n-1)
    print(n,"the sum is",t)
    return n+sum(n-1)

sum(10)
```

3 Exception Handling

3.1 Error Handling

Explanation

People are not saints who never make mistakes. When writing programs, it is often challenging to ensure that all of our code is accurate the first time we write it. Therefore, we need to take measures well to handle errors and avoid dangerous situations. Thus, we need to learn how to deal with mistakes.

The error handling discussed here includes errors, exceptions, etc., which are considered abnormal situations encountered in the program. They are not the same as the expected normal execution flow. These situations may be due to problems with the logic of the program itself, for example, criminal division by zero, array overruns, etc.; they may also be due to temporary unavailability of resources, such as network requests, memory requests, etc.; there may also be direct hardware problems, sudden power failures, server crashes, etc. It is difficult to avoid the hardware situation, we mainly think about how to avoid the error in the case of running code.

Exercise

Can you think of any of those ways of dealing with errors, and if so, please give examples?

Solution

- Special judgments are used to avoid errors in advance and allow only the correct data to pass.
- If there is an error in the program, the error is fed back to the relevant person so that the error can be pointed out.

- Set up as many fault tolerance mechanisms as possible to avoid errors.

3.2 Defensive Programming

Explanation

In programming, we are often faced with a large amount of code that usually takes a long time to maintain, and this requires a lot of workforces and a lot of time. To reduce the cost after the program is completed, we need to write the code more concisely and more standardized. There are often many errors in programming. These errors are often very hidden, especially when the amount of code is very large. It will be tough to find them, and a mistake can lead to dire consequences. This requires us to be very careful when writing code. Thus, in large coding projects, we need to use defensive programming.

Based on the need to build the pre-engineering and maintain the code later, we need to make the code more concise, understandable, and robust. To prevent unpredictable failures, we often need to consider more situations, such as the variety of user input and possible attacks from hackers, to write more high-quality code.

In order to achieve the above requirements, we can do the following:

- Asking Permission
- Asking Forgiveness

We will explain how to do this in the following topics.

Exercise

Why do we need code defensively?

Solution

- Ensure the correctness and readability of the program.
- Ensure the robustness and stability of the program in the future.
- Make it easier for someone to pick up the code.

3.3 Asking Permission - Raising Errors

Explanation

When we program, we often need to consider unpredictable inputs in the future. So, before we know what the user will type, it's good to anticipate the information in advance and handle certain illegitimate cases in case our program crashes. For each statement in the program, before executing it, we can consider the situation of completing this statement and handling certain illegal situations. For example, when buying how many items, it is evident that

the number cannot be harmful. The input string must be a number and not some other meaningless character in the input.

Specifically, in Python, we can get permission to run the code by using the *if* judgment statement. More specifically, for each input from the user, we need first to use the *isinstance* to determine whether the value entered by the user is what we expect, such as whether it is a number, etc. Only after determining that the input is a number can we make different judgments about the information, such as using logical operators to determine if the number is less than zero. These operations can help us avoid program errors if the user input is unknown, thus reducing the loss significantly. When these errors occur, we can process these errors by raising error.

In a word, we can use common sense to determine many cases of discordance and write these discordance cases into the program judgment conditions. Thus we can increase the robustness of the program.

When these errors occur, we can process these errors by raising an error. It is better to give more apparent hints for these errors so that the mistakes can be corrected later.

Exercise

Enter the price of an item and the quantity purchased, and output the total cost. If an error is entered, output a prompt message.

Solution

In order to avoid input errors, we need to determine the type and size of the input, and for incorrect information, we process it by raise.

```
1 def buy(price, number):
2     "calculate the total cost"
3     if not instance(price, int) or not instance(number, int):
4         raise TypeError("illegal input,prict or number should be an
              integer")
5     if price <= 0 or number <=0:
6         raise ValueError("negative value,prict or number should be
              positive")
7     return price *number
```

3.4 Asking Permission - Repetition

Explanation

Above, we focused on determining whether the function is correct but neglected to determine whether the user's initial input is accurate. If the user's input is incorrect, and this happens frequently, how should we deal with it? We can solve this kind of error by a circular prompt; for example, when input prompts the user for input, we first initialize an illegal value and determine if

the input is legal. We first enter the first loop since the initial value is not permitted. In the first loop, we need the user to make the first input, and after the input, we first determine whether the user's input is the type of input we need, and if it is, we update the current value. If not, we will still keep the illegal deal. After transforming the user's input, we find that our input is still not legal; we will use the while loop to go to the next circle level. We can jump out of the loop and continue with the following operations if it is already legal. This way, we can ensure that the user's input is correct.

Exercise

The salesperson enters the price of an item and the number of articles and determines whether the input is legal.

Solution

```
def main():

    # get two integers from the salesperson
    price = -1
    while price < 0:
        price_text = input("how much is the item?")
        if price_text.isnumeric():
            price = int(price_text)
    # get the number of the item
    number = -1
    while number < 0:
        number_text = input("how many is the item")
        if number_text.isnumeric():
            number = int(number_text)
```

3.5 Asking Forgiveness

Explanation

Although we have the error avoidance methods described above, we need more specific error avoidance methods to reinforce our defensive programming. Ask forgiveness is one of these methods.

Asking forgiveness relies on the use of "try-except" to determine errors. In the past, we were able to display errors but needed to nest multiple if statements to show the specific error. Now, we can use the "try-except" statement to scout for mistakes and output the errors in categories. This brings us a lot of conveniences, such as no longer having to write cumbersome reports and not always being afraid to be thoughtful.

We can handle a single exception with this statement, as shown below.

```
>>> try:
>>>     print(5/0)
>>> except ZeroDivisionError as e:
>>>     print(e)
```

It is also possible to handle multiple exceptions separately.

```
>>> a = [0,1]
>>> try:
>>>     print(a[3])
>>>     print(5/0)
>>> except ZeroDivisionError as e:
>>>     print(e)
>>> except IndexError as e:
>>>     print(e)
list index out of range
```

Multiple exceptions can also be handled in a unified manner:

```
>>> try:
>>>     code
>>> except (Error1,Error) :
>>>     print(e)
```

Exercise

Using Asking Forgiveness, realize the purchase price and purchase quantity for e.g. items, and output the total price.

Solution

```
def buy(price, number):
    "calculate the total cost"
    if not instance(price, int) or not instance(number, int):
        raise TypeError("illegal input,prict or number should be an
            integer")
    if price <= 0 or number <=0:
        raise ValueError("negative value,prict or number should be
            positive")
    return price *number

def main():
    try:
    # get two integers from the salesperson
        price_text = input("how much is the item?")
        price = int(price_text)
    # get the number of the item
```

```
        number_text = input("how many is the item")
        number = int(number_text)
        print(buy(price,number))
    except TypeError as ex:
        print("Invalid type",ex)
    except ValueError as ex:
        print("ValueError",ex)
    except Exception as ex:
        print(ex)
```

3.6 Understanding Execution Paths

Explanation

Depending on the different inputs and errors, we then have different program execution paths. We execute the program in regular order according to the user's information. When encounter a judgment statement,we may choose a different run path according to the judgment statement. If there an error occurred, the program will be terminated immediately. But if this code is surrounded by catch, the net will catch the mistake and compare it with the error to the following exception. If the mistake matches successfully, it will go to the code under that error for execution and end the program after the performance is finished.

Exercise

1. Point out the output of the following program, if the input is "five"

```
try:
    number = int(input())
    print(number)
except TypeError:
    print("Type error")
except ValueError:
    print("Value error")
```

2. When "-1" is entered, how will the following code be executed?

```
try:
    number = int(input())
    f(number)      # IndexError is raised
except TypeError:
    print("Type error")
except ValueError:
    print("Value error")
```

Solution

1. "Value error". Because the string "five" is not the value which can be converted to an integer and it raises a ValueError.

2. Program crashes. Because the f() function will raise an error the code won't catch.

3.7 Files

Explanation

In our work, we often need to refer to content in other files. In python, we can use the open function to meet such a need.

Python provides the necessary functions and methods to perform basic file operations by default. We can use the file object for most file operations. For example, reading and writing. We must first open a file with Python's built-in open() function to create a file object before the relevant methods can be called to read or write to it. The syntax is shown below.

```
file object = open(file_name [, access_mode][, buffering])
```

The details of each parameter are as follows:

- file_name: The file_name variable is a string value containing the path of the file you want to access.

- access_mode: access_mode determines the mode of opening the file: read-only, write, append, etc. This parameter is not mandatory and the default file access mode is read-only (r).

- buffering: If the value of buffering is set to 0, there will be no hosting. If the value of buffering is set to 1, accessing the file will host the lines. If the value of buffering is set to an integer greater than 1, it indicates that this is the buffer size of the hosting area. If the value is negative, the buffering size is the system default.

After a file is opened, you have a file object, you can get various information about the file and do what you want with the file object according to the open mode. Here are the functions we can call.

function	description
write	The write() method can write any string to an open file. It is important to note that Python strings can be binary data, not just text and write() method does not add a line break to the end of the string.
read	The read() method reads a string from an open file.
close	The close() method of the File object flushes the buffer of any information that hasn't been written and closes the file, after which no more writes can be made. Python closes the previous file when a reference to a file object is reassigned to another file. It is a good habit to close files with the close() method.

Table 1: Caption

Of course we can also use a for loop to read in line by line. Also, there are several exceptions we need to be aware of when reading and writing files.

- FileNotFoundError: the file doesn't exist
- PermissionError: there is no permission
- OSError: there is some error with the IO

Exercise

Read the file 1.txt, convert all lowercase letters in it to uppercase letters, and then write the processed text to 2.txt.

Solution

```
def main():
    in_text = open("1.txt","r")
    out_text = open("2.txt","w")

    for line in in_text:
        out_text.write(line.upper())

    in_text.close()
    out_text.close()
```

3.8 Collecting Data

Explanation

When writing the program, we try to consider all possible cases of inputs, and for each input, discard the ones that do not meet our requirements and keep

the ones that meet our needs. We need to synthesize and apply the centralized approach illustrated above to achieve the requirements.

The loop allows us to continuously collect the input, discard the unwanted information and remind the user of the correct input by judging it, and terminate the loop by setting the exit condition. Also when we read and write files, we need to consider the path of the file, whether the file already exists, whether it overwrites the file, and other different environmental situations. Through layers of if loops we can determine whether the user's input conforms to the specification. And avoid program crash by catching exceptions.

Exercise

Write a program that reads in the students' scores and stores them in "score.text". Pay attention to various special cases.

Solution

```
"""
Module: Handing Exceptions

This file implements the reading of score from the keyboard
and storing them in a file, one per line
"""
import os

def main():

    try:
        filename = input("Enter filename: ")
        if os.path.isdir(filename):
            print("that is not a file")
        elif os.path.exists(filename):
            answer = input("Do you want to replace the file (y/n): ")
            if answer == "y":
                file = open(filename, "w")
            else:
                file = open(filename, "a")
                number = 0
                while number != -1:
                    try:
                        number = int(input("Enter score (or -1 to quit):
                            "))
                        if number > 0:
                            file.write(str(number))
                            file.write("\n")

                    except ValueError as ex:
                        print("Enter an integer value.")
```

```
31
32                  file.close()
33      except PermissionError:
34          print("You do not have permission to use that file")
35      except OSError:
36          print("Something happened while writing to the file:", filename)
37
38  main()
```

4 Classes and Objects

4.1 Classes and Objects

Explanation

Early computer programming was based on a process-oriented approach, such as implementing the arithmetic operation 1+1+2 = 4 by designing an algorithm to solve the day's problem. As computer technology continued to improve, computers were used to solve increasingly complex problems. Everything is an object, and real-world things are abstracted into objects through the object-oriented approach. Real-world relationships are summarized into classes and inheritance to help people achieve abstraction and digital modeling of the real world. The object-oriented approach is more conducive to analyzing, designing, and programming complex systems in a human-understood way. At the same time, an object-oriented can effectively improve the efficiency of programming; through encapsulation techniques, message mechanisms can be like building blocks to develop a brand new system quickly. Object-oriented refers to a programming paradigm but also a method of program development. An object is a concrete implementation of a class. It takes objects as the basic unit of a program and encapsulates programs and data to improve software's reusability, flexibility, and extensibility.

Exercise

Take a life example and explain why we need classes and objects.

Solution

For a school student management system, no matter it is simple or complex, it is always implemented around the two objects of students and teachers. In nature, every object has some attributes and behaviors, such as students have the student number, name, gender and other attributes, as well as exams, experiments and other behaviors. Therefore, each individual can be described in terms of attributes and behaviors.

4.2 Intro to Classes

Explanation

Classes are the basis for object-oriented programming (OOP) to achieve information encapsulation. A class is a user-defined reference data type, also known as a class type. Each class contains a description of the data and a set of functions that manipulate the data or pass messages, and instances of classes are called objects.

A class is an abstraction of a course of real-life things that have common characteristics. If the data types provided in a program directly correspond to the concepts in the application, the program will be easier to understand and easier to modify. A well-chosen set of user-defined classes will make the program more concise. In addition, it makes various forms of code analysis easier to perform. In particular, it makes it possible for the compiler to check for illegal use of objects.

The essence of a class is a reference data type, similar to basic data types such as int, float, double, etc. The difference is that it is a complex data type. Because it is essentially a data type and not data, it does not exist in memory and cannot be manipulated directly; it only becomes manipulable when instantiated as an object.

Classes encapsulate internal properties and methods for manipulating their owners. A class is a definition of some object with behavior that describes what an object can do and how it does it, and they are the procedures and processes that can be performed on that object. It contains information about how the thing behaves, including its name, properties, methods, and events.

The composition of a class includes member attributes and member methods. Data members correspond to the properties of a course, and data members of a class are also a data type and do not require memory allocation. The member functions are used to manipulate the properties of the course and are specific to a class; for example, a "student" can "attend a class," but a "fruit" cannot. The operations of a class that interacts with the outside world are called interfaces.

Exercise

Determine the correctness or incorrectness of the following statements, and if no, explain why.

1. A class is an abstract collection of concrete things that can reflect a thing as a whole.

2. All operations inside a class are called functions, and all variables are called attributes.

3. In order to describe various kinds of buildings, we abstract these things as Building class.

4. To describe buildings, we abstract these buildings into a particular building class.

Solution

1. True
2. False. Operations in classes are not called functions, but methods.
3. True. A class is an abstract sum of external things.
4. False. Using a concrete thing as a class for an abstract thing is not allowed.

4.3 Working with Objects

Explanation

Once we have a class, we can instantiate an object and build it through Constructor. We pass in the variables needed for the constructor, and then the class calls the constructor and returns an object to us. This object has been allocated memory and becomes a real data structure. We can now access the attributes in this object and call the relevant methods in the object, depending on our specific needs.

In order to access the attributes and methods in the object, we need to use dot-notation to do so. To access the attribute,we can write the code like this:

```
object.attribute
```

To access the method in the object, we can write the code like the following:

```
object.method()
```

Exercise

Given a building class, the constructor needs to pass in the name of the building and the type of building, the internal class attribute has height, etc., and the internal class methods have buy and sell methods.

Instantiate a structure and do whatever you want to do.

Solution

```
from building import Building

def main():
    my_house = Building(" Big House","House")
    print("Height: ",my_house.height)

    you_house = Building("Big Theater", "Theater")
    print("Height: ",you_house.height)

    my_house.sell()
```

```
11      you_house.buy()
```

4.4 Defining Classes

Explanation

To build a class, we first need to specify the object. We need to abstract the things we need to deal with, extract the properties and methods inherent to such items, and thus design the internal structure of a class. Once we have the internal network, we can construct the course. We declare the properties and methods we need and implement them separately from the original class internal structure. In addition, it is essential to write the constructor.

We need to initialize the constructor __init__(). __init__() is a unique method that Python automatically runs whenever you create a new instance based on a class. The name of this method has two underscores at the beginning and two at the end, which is a convention designed to avoid name conflicts between Python's default methods and standard methods.

We can define multiple formal parameters in the __init__() method. However, in this method definition, the standard parameter self is essential and must also precede the other common parameters. Why is it necessary to include the traditional parameter self in the method definition? Because when Python calls this __init__() method to create an instance, the actual parameter self is automatically passed. Call automatically passes the authentic parameter self, which refers to the instance itself, giving instant access to the class's properties and methods.

Variables prefixed with self are available to all methods in the class, and we can also access these variables through any instance of the class. For example, self.name = name gets the attribute stored in the instance and stores the value, which is associated with the currently created instance. Other statements works similarly.

Exercise

Create an employee class, which records the name and salary of the employee and allows you to count the number of employees. In this class, you can display the number of all employees and display the current employee's name and salary information.

Solution

```
1   class Employee:
2      empCount = 0
3
4      def __init__(self, name, salary):
5         self.name = name
```

```
6         self.salary = salary
7         Employee.empCount += 1
8
9     def displayCount(self):
10      print "Total Employee %d" % Employee.empCount
11
12    def displayEmployee(self):
13       print "Name : ", self.name, ", Salary: ", self.salary
```

4.5 Constructors

Explanation

To create an object, we need to define the constructor __init__() method. The constructor method is used to perform the "initialization of the instance object," i.e., after the object is created, it initializes the current object's properties and has no return value.

The main points of __init__() are as follows.

- The name is fixed, it must be __init__()
- the first parameter is fixed, must be self. self refers to the sample object just created.
- constructor is usually used to initialize the sample properties, the following code is to initialize the sample properties.

```
1  class Student:
2       def __init__(self,name,age):
3          self.name = name
4          slef.age = age
```

- The constructor is called by the class name (parameter list), and after the call, the created object is returned to the corresponding variable.
- __init__() method: initialize the created object, initialization means: "assign a value to the instance property"
- __new__() method: used to create the object, but we generally do not need to define this method.

Also in the initialization method, we need to follow the rules for passing parameters to the function, as we learned before.

Exercise

1. In order to initialize the object, we must call the constructor method to initialize the object.

2. Select the correct option for the following constructor call in the class Employee.

```
def __init__(self, name,salary = 0):
```

A tom = Employee("Tom")

B tom = Employee()

C tom = Employee("Tom",10)

D tom = Employee("Tom",salary = 10)

Solution

1. Yes. The constructor must be called when the object is instantiated.
2. ACD. According to the rules for passing references to functions, variables without initial values are necessary to be assigned values.So the B is uncorrect.

4.6 Printing Objects

Explanation

In our programming process, if we want to see all the variables and functions in a particular object, we need to do a lot of printing operations to reveal the internals of a specific class. We would like to have some way to make our printing easier yet, support direct printing using print(). Python's classes provide the method __str__() method to meet our needs.

In method __str__(), we can print it's returned value out directly using method print() by returning a string describing the class and object. Because calling print() in python will call __str__() to print the instantiated object, if __str__() has a return value, it will print the return value of the method; if not, it will print the address and other information of the object by default.

Exercise

Write a child class that contains information such as name, gender, etc., and requires an instance object to be able to print out the string like "Jack is a boy" when passed into the print function.

Solution

```
class Boy:

    def __init__(self, name, sex):
        self.name = name
        self.sex = sex
```

```
6
7       def __str__(self):
8           output = self.name + " is a " + self.sex
9           return output
10
11
12  boy = Boy("Jack", "boy")
13  print(boy)
```

4.7 Testing Classes

Explanation

Once we have written an object, we need to test the internal components of the object, which requires us to use unit tests. Unit testing refers to the inspection and verification of the software's smallest testable unit (a module, a function, or a class).

unittest is Python's built-in standard class library to create a test case by inheriting *unittest*. Inheritance allows us to be able to use the variables and methods of the parent class, thus reducing redundant code writing and increasing the efficiency of writing code.

In library *unittest*, we can test our class using the methods in the following table.

method	description
assertEqual	If the two input values are equal, then assertEqual() will return true, otherwise it will return false.
assertNotEqual	If the two input values are not equal, then assertEqual() will return true, otherwise it will return false.
assertTrue	If the input value is true, assertTrue() will return true, otherwise it will return false.
assertFalse	If the input value is false,the method will return true, otherwise it will return false.
assertAlmostEqual	If the two input float objects are almost equal,the method will return true,otherwise it will return false.

Table 2: unittest methods

With these functions in place, we can determine if our code is correct by testing the results in the object. If the code is wrong, we can locate our error more accurately based on the error message.

Exercise

Try to test the following class for the relevant functions and give the code.

```
class House:

    ''' class: House
    Attributes: height, owner, price
    Methods: add a feature (changes the price)
    '''


    def __init__(self, height, owner, price=25000.0):
        '''
        Constructor -- creates an new instance of a building
        Parameters:
        self -- the current object
        height -- the initial height of this house
        owner -- the current owner of this house
        price (optional) -- the initial price of this house
        '''
        self.height = height
        self.owner = owner
        self.price = price


    def add_feature(self, item):


        '''
        Method -- add a feature to this house
        Parameters:
        self -- the current object
        item -- the feature to add to this house
        Returns nothing
        '''
        if item == "TV":
            self.price += 200
        elif item == "Layer":
            self.price += 350
            self.height += 20
```

Solution

In the class House, we need to check the price and the height to tell whether our code is right.

```
from  wu import House
import unittest

class Test(unittest.TestCase):
    def test_init(self):
```

```
        house = House(20,"Jack")
        self.assertEqual(house.height,20)
        self.assertEqual(house.owner,"Jack")
        self.assertAlmostEqual(house.price,25000.0)

    def test_add_feature(self):
        house = House(20,"Jack")
        house.add_feature("TV")
        self.assertAlmostEqual(house.price,25200.0)

        house.add_feature("Layer")
        self.assertAlmostEqual(house.price,25350.0)
        self.assertEqual(house.height,40)
```