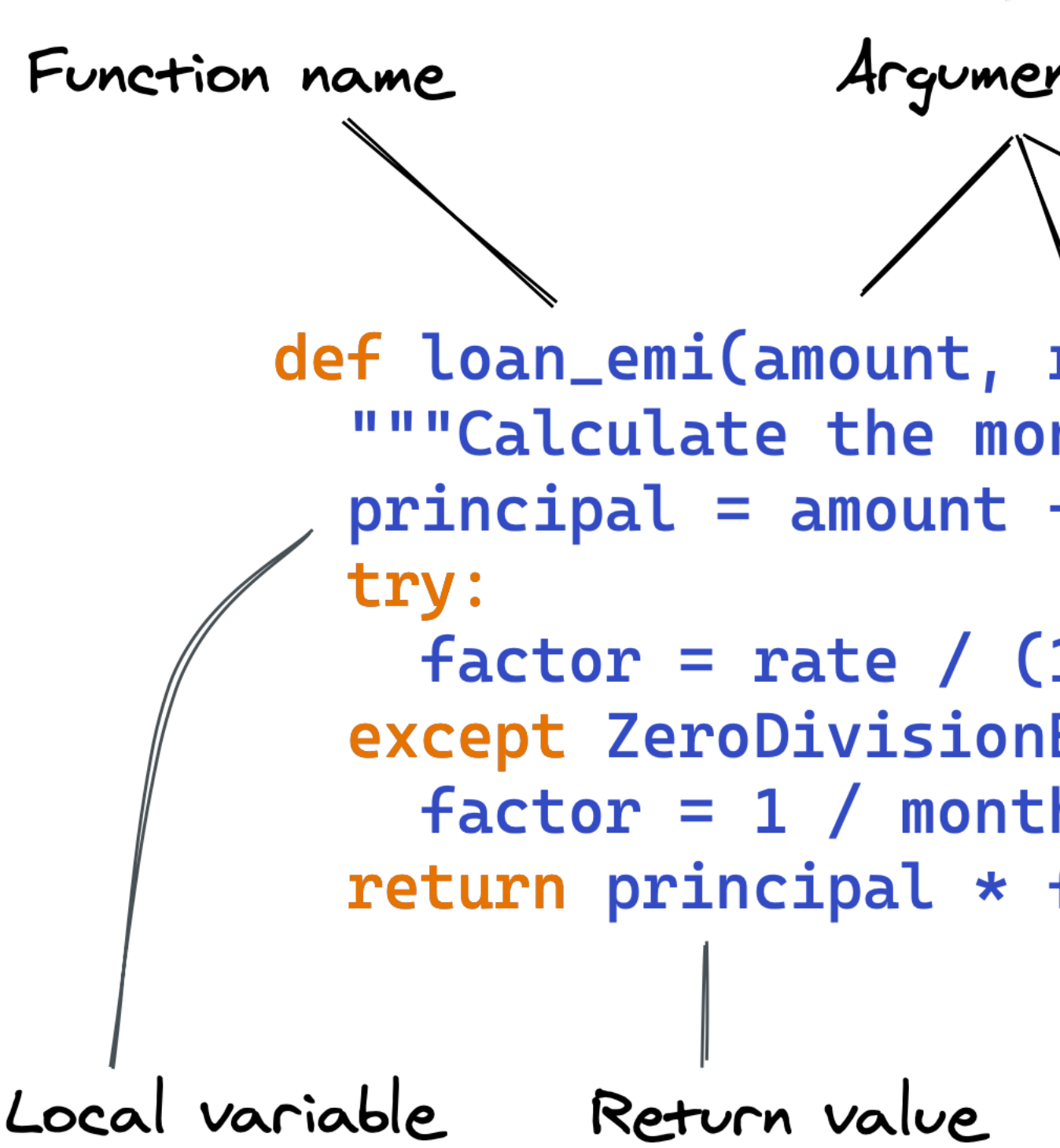
Writing Reusable Code using Functions in Python

This tutorial is a part of Data Analysis with Python: Zero to Pandas and Zero to Data Analyst Science Bootcamp.



These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself.

This tutorial covers the following topics:

- Creating and using functions in Python
- Local variables, return values, and optional arguments
- Reusing functions and using Python library functions
- Exception handling using `try-except` blocks
- Documenting functions using docstrings

Creating and using functions

A function is a reusable set of instructions that takes one or more inputs, performs some operations, and often returns an output. Python contains many in-built functions like `print`, `len`, etc., and provides the ability to define new ones.

In [ ]:
```
today = "Saturday"
print("Today is", today)
```

```
Today is Saturday
```

You can define a new function using the `def` keyword.

In [ ]:
```
def say_hello():
    print('Hello there!')
    print('How are you?')
```

Note the round brackets or parentheses `()` and colon `:` after the function's name. Both are essential parts of the syntax. The function's *body* contains an indented block of statements. The statements inside a function's body are not executed when the function is defined. To execute the statements, we need to *call* or *invoke* the function.

In [ ]:
```
say_hello()
```

```
Hello there!
How are you?
```

Function arguments

Functions can accept zero or more values as *inputs* (also knows as *arguments* or *parameters*). Arguments help us write flexible functions that can perform the same operations on different values. Further, functions can return a result that can be stored in a variable or used in other expressions.

Here's a function that filters out the even numbers from a list and returns a new list using the `return` keyword.

In [ ]:
```
def filter_even(number_list):
    result_list = []
    for number in number_list:
        if number % 2 == 0:
            result_list.append(number)
    return result_list
```

Can you understand what the function does by looking at the code? If not, try executing each line of the function's body separately within a code cell with an actual list of numbers in place of `number_list`.

In [ ]:
```
even_list = filter_even([1, 2, 3, 4, 5, 6, 7])
```

In [ ]:
```
even_list
```

Out [6]:
```
[2, 4, 6]
```

Writing great functions in Python

As a programmer, you will spend most of your time writing and using functions. Python offers many features to make your functions powerful and flexible. Let's explore some of these by solving a problem:

> Radha is planning to buy a house that costs `$1,260,000`. She considering two options to finance her purchase:
>
> - Option 1: Make an immediate down payment of `$300,000`, and take loan 8-year loan with an interest rate of 10% (compounded monthly) for the remaining amount.
> - Option 2: Take a 10-year loan with an interest rate of 8% (compounded monthly) for the entire amount.
>
> Both these loans have to be paid back in equal monthly installments (EMIs). Which loan has a lower EMI among the two?

Since we need to compare the EMIs for two loan options, defining a function to calculate the EMI for a loan would be a great idea. The inputs to the function would be cost of the house, the down payment, duration of the loan, rate of interest etc. We'll build this function step by step.

First, let's write a simple function that calculates the EMI on the entire cost of the house, assuming that the loan must be paid back in one year, and there is no interest or down payment.

In [ ]:
```
def loan_emi(amount):
    emi = amount / 12
    print('The EMI is ${}'.format(emi))
```

In [ ]:
```
loan_emi(1260000)
```

```
The EMI is $105000.0
```

Local variables and scope

Let's add a second argument to account for the duration of the loan in months.

In [ ]:
```
def loan_emi(amount, duration):
    emi = amount / duration
    print('The EMI is ${}'.format(emi))
```

Note that the variable `emi` defined inside the function is not accessible outside. The same is true for the parameters `amount` and `duration`. These are all *local variables* that lie within the *scope* of the function.

> **Scope**: Scope refers to the region within the code where a particular variable is visible. Every function (or class definition) defines a scope within Python. Variables defined in this scope are called *local variables*. Variables that are available everywhere are called *global variables*.

Scope rules allow you to use the same variable names in different functions without sharing values from one to the other.

In [ ]:
```
emi
```

```
---------------------------------------------------------------------------NameError                                 Traceback (most recent call las
----> 1 emi
NameError: name 'emi' is not defined
```

In [ ]:
```
amount
```

```
---------------------------------------------------------------------------NameError                                 Traceback (most recent call las
----> 1 amount
NameError: name 'amount' is not defined
```

In [ ]:
```
duration
```

```
---------------------------------------------------------------------------NameError                                 Traceback (most recent call las
----> 1 duration
NameError: name 'duration' is not defined
```

We can now compare a 6-year loan vs. a 10-year loan (assuming no down payment or interest).

In [ ]:
```
loan_emi(1260000, 8*12)
```

```
The EMI is $13125.0
```

In [ ]:
```
loan_emi(1260000, 10*12)
```

```
The EMI is $10500.0
```

Return values

As you might expect, the EMI for the 6-year loan is higher compared to the 10-year loan. Right now, we're printing out the result. It would be better to return it and store the results in variables for easier comparison. We can do this using the `return` statement

In [ ]:
```
def loan_emi(amount, duration):
    emi = amount / duration
    return emi
```

In [ ]:
```
emi1 = loan_emi(1260000, 8*12)
```

In [ ]:
```
emi2 = loan_emi(1260000, 10*12)
```

In [ ]:
```
emi1
```

Out [18]:
```
13125.0
```

In [ ]:
```
emi2
```

Out [19]:
```
10500.0
```

Optional arguments

Next, let's add another argument to account for the immediate down payment. We'll make this an *optional argument* with a default value of 0.

In [ ]:
```
def loan_emi(amount, duration, down_payment=0):
    loan_amount = amount - down_payment
    emi = loan_amount / duration
    return emi
```

In [ ]:
```
emi1 = loan_emi(1260000, 8*12, 3e5)
```

In [ ]:
```
emi1
```

Out [22]:
```
10000.0
```

In [ ]:
```
emi2 = loan_emi(1260000, 10*12)
```

In [ ]:
```
emi2
```

Out [24]:
```
10500.0
```

Next, let's add the interest calculation into the function. Here's the formula used to calculate the EMI for a loan:

$$EMI = \frac{P \times r \times (1+r)^n}{(1+r)^n - 1}$$

where:

- `P` is the loan amount (principal)
- `n` is the no. of months
- `r` is the rate of interest per month

The derivation of this formula is beyond the scope of this tutorial. See this video for an explanation: https://youtu.be/Coxza9ugW4E .

In [ ]:
```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    return emi
```

Note that while defining the function, required arguments like `cost`, `duration` and `rate` must appear before optional arguments like `down_payment`.

Let's calculate the EMI for Option 1

In [ ]:
```
loan_emi(1260000, 8*12, 0.1/12, 3e5)
```

Out [26]:
```
14567.19753389219
```

While calculating the EMI for Option 2, we need not include the `down_payment` argument.

In [ ]:
```
loan_emi(1260000, 10*12, 0.08/12)
```

Out [27]:
```
15287.276888775077
```

Named arguments

Invoking a function with many arguments can often get confusing and is prone to human errors. Python provides the option of invoking functions with *named* arguments for better clarity. You can also split function invocation into multiple lines.

In [ ]:
```
emi1 = loan_emi(
    amount=1260000,
    duration=8*12,
    rate=0.1/12,
    down_payment=3e5
)
```

In [ ]:
```
emi1
```

Out [29]:
```
14567.19753389219
```

In [ ]:
```
emi2 = loan_emi(amount=1260000, duration=10*12, rate=0.08/12)
```

In [ ]:
```
emi2
```

Out [31]:
```
15287.276888775077
```

Modules and library functions

We can already see that the EMI for Option 1 is lower than the EMI for Option 2. However, it would be nice to round up the amount to full dollars, rather than showing digits after the decimal. To achieve this, we might want to write a function that can take a number and round it up to the next integer (e.g., 1.2 is rounded up to 2). That would be a great exercise to try out!

However, since rounding numbers is a fairly common operation, Python provides a function for it (along with thousands of other functions) as part of the Python Standard Library. Functions are organized into *modules* that need to be imported to use the functions they contain.

> **Modules**: Modules are files containing Python code (variables, functions, classes, etc.). They provide a way of organizing the code for large Python projects into files and folders. The key benefit of using modules is *namespaces*: you must import the module to use its functions within a Python script or notebook. Namespaces provide encapsulation and avoid naming conflicts between your code and a module or across modules.

We can use the `ceil` function (short for *ceiling*) from the `math` module to round up numbers. Let's import the module and use it to round up the number `1.2`.

In [ ]:
```
import math
```

In [ ]:
```
help(math.ceil)
```

```
Help on built-in function ceil in module math:

ceil(x, /)
    Return the ceiling of x as an Integral.

    This is the smallest integer >= x.
```

In [ ]:
```
math.ceil(1.2)
```

Out [34]:
```
2
```

Let's now use the `math.ceil` function within the `home_loan_emi` function to round up the EMI amount.

Using functions to build other functions is a great way to reuse code and implement complex business logic while still keeping the code small, understandable, and manageable. Ideally, a function should do one thing and one thing only. If you find yourself writing a function that does too many things, consider splitting it into multiple smaller, independent functions. As a rule of thumb, try to limit your functions to 10 lines of code or less. Good programmers always write short, simple, and readable functions.

In [ ]:
```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    emi = math.ceil(emi)
    return emi
```

In [ ]:
```
emi1 = loan_emi(
    amount=1260000,
    duration=8*12,
    rate=0.1/12,
    down_payment=3e5
)
```

In [ ]:
```
emi1
```

Out [37]: 14568

In [ ]:
```
emi2 = loan_emi(amount=1260000, duration=10*12, rate=0.08/12)
```

In [ ]:
```
emi2
```

Out [39]: 15288

Let's compare the EMIs and display a message for the option with the lower EMI.

In [ ]:
```
if emi1 < emi2:
    print("Option 1 has the lower EMI: ${}".format(emi1))
else:
    print("Option 2 has the lower EMI: ${}".format(emi2))
```

```
Option 1 has the lower EMI: $14568
```

Reusing and improving functions

Now we know for sure that "Option 1" has the lower EMI among the two options. But what's even better is that we now have a handy function `loan_emi` that we can use to solve many other similar problems with just a few lines of code. Let's try it with a couple more questions.

> **Q**: Shaun is currently paying back a home loan for a house he bought a few years ago. The cost of the house was <math xmlns="http://www.w3.org/1998/Math/MathML"><semantics><mrow><mn>800</mn><mo separator="true">,</mo><mn>000</mn><mi mathvariant="normal">'</mi><mi mathvariant="normal">.</mi><mi>S</mi><mi>h</mi><mi>a</mi><mi>u</mi><mi>n</mi><mi>m</mi><mi>a</mi><mi>d</mi><mi>e</mi><mi>a</mi><mi>d</mi><mi>o</mi><mi>w</mi><mi>n</mi><mi>p</mi><mi>a</mi><mi>y</mi><mi>m</mi><mi>e</mi><mi>n</mi><mi>t</mi><mi>o</mi><mi>f</mi><mi mathvariant="normal">'</mi><mn>25</mn></mrow><annotation encoding="application/x-tex">800,000</annotation></semantics></math>800,000'.Shaunmadeadownpaymentof'2560,000, which he is planning to finance using a 1-year loan with an interest rate of 12% per annum. Both loans are paid back in EMIs. What is the total monthly payment Shaun makes towards loan repayment?

This question is now straightforward to solve, using the `loan_emi` function we've already defined.

In [ ]:
```
cost_of_house = 800000
home_loan_duration = 6*12 # months
home_loan_rate = 0.07/12 # monthly
home_down_payment = .25 * 800000

emi_house = loan_emi(amount=cost_of_house,
                     duration=home_loan_duration,
                     rate=home_loan_rate,
                     down_payment=home_down_payment)

emi_house
```

Out [41]: 10230

In [ ]:
```
cost_of_car = 60000
car_loan_duration = 1*12 # months
car_loan_rate = .12/12 # monthly

emi_car = loan_emi(amount=cost_of_car,
                   duration=car_loan_duration,
                   rate=car_loan_rate)

emi_car
```

Out [42]: 5331

In [ ]:
```
print("Shaun makes a total monthly payment of ${} towards loan repayments.".format(emi_house+emi_car))
```

```
Shaun makes a total monthly payment of $15561 towards loan repayments.
```

Exceptions and `try-except`

> Q: If you borrow `$100,000` using a 10-year loan with an interest rate of 9% per annum, what is the total amount you end up paying as interest?

One way to solve this problem is to compare the EMIs for two loans: one with the given rate of interest and another with a 0% rate of interest. The total interest paid is then simply the sum of monthly differences over the duration of the loan.

In [ ]:
```
emi_with_interest = loan_emi(amount=100000, duration=10*12, rate=0.09/12)
emi_with_interest
```

Out [44]: 1267

In [ ]:
```
emi_without_interest = loan_emi(amount=100000, duration=10*12, rate=0./12)
emi_without_interest
```

```
---------------------------------------------------------------------------ZeroDivisionError                         Traceback (most recent call las
----> 1 emi_without_interest = loan_emi(amount=100000, duration=10*12, rate=0./12)
      2 emi_without_interest
<ipython-input-35-ad16168becb0> in loan_emi(amount, duration, rate, down_payment)
      1 def loan_emi(amount, duration, rate, down_payment=0):
      2     loan_amount = amount - down_payment
----> 3     emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
      4     emi = math.ceil(emi)
      5     return emi
ZeroDivisionError: float division by zero
```

Something seems to have gone wrong! If you look at the error message above carefully, Python tells us precisely what is wrong. Python *throws* a `ZeroDivisionError` with a message indicating that we're trying to divide a number by zero. `ZeroDivisonError` is an *exception* that stops further execution of the program.

> **Exception**: Even if a statement or expression is syntactically correct, it may cause an error when the Python interpreter tries to execute it. Errors detected during execution are called exceptions. Exceptions typically stop further execution of the program unless handled within the program using `try-except` statements.

Python provides many built-in exceptions *thrown* when built-in operators, functions, or methods are used incorrectly: https://docs.python.org/3/library/exceptions.html#built-in-exceptions. You can also define your custom exception by extending the `Exception` class (more on that later).

You can use the `try` and `except` statements to *handle* an exception. Here's an example:

In [ ]:
```
try:
    print("Now computing the result..")
    result = 5 / 0
    print("Computation was completed successfully")
except ZeroDivisionError:
    print("Failed to compute result because you were trying to divide by zero")
    result = None

print(result)
```

```
Now computing the result..
Failed to compute result because you were trying to divide by zero
None
```

When an exception occurs inside a `try` block, the block's remaining statements are skipped. The `except` block is executed if the type of exception thrown matches that of the exception being handled. After executing the `except` block, the program execution returns to the normal flow.

You can also handle more than one type of exception using multiple `except` statements. Learn more about exceptions here: https://www.w3schools.com/python/python_try_except.asp .

Let's enhance the `loan_emi` function to use `try-except` to handle the scenario where the interest rate is 0%. It's common practice to make changes/enhancements to functions over time as new scenarios and use cases come up. It makes functions more robust & versatile.

In [ ]:
```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    try:
        emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    except ZeroDivisionError:
        emi = loan_amount / duration
```

```
    emi = math.ceil(emi)
    return emi
```

We can use the updated `loan_emi` function to solve our problem.

> **Q**: If you borrow `$100,000` using a 10-year loan with an interest rate of 9% per annum, what is the total amount you end up paying as interest?

In [ ]:
```
emi_with_interest = loan_emi(amount=100000, duration=10*12, rate=0.09/12)
emi_with_interest
```

Out [48]: 1267

In [ ]:
```
emi_without_interest = loan_emi(amount=100000, duration=10*12, rate=0)
emi_without_interest
```

Out [49]: 834

In [ ]:
```
total_interest = (emi_with_interest - emi_without_interest) * 10*12
```

In [ ]:
```
print("The total interest paid is ${}.".format(total_interest))
```

```
The total interest paid is $51960.
```

Documenting functions using Docstrings

We can add some documentation within our function using a *docstring*. A docstring is simply a string that appears as the first statement within the function body, and is used by the `help` function. A good docstring describes what the function does, and provides some explanation about the arguments.

In [ ]:
```
def loan_emi(amount, duration, rate, down_payment=0):
    """Calculates the equal montly installment (EMI) for a loan.

    Arguments:
        amount - Total amount to be spent (loan + down payment)
        duration - Duration of the loan (in months)
        rate - Rate of interest (monthly)
        down_payment (optional) - Optional intial payment (deducted from amount)
    """
    loan_amount = amount - down_payment
    try:
        emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    except ZeroDivisionError:
        emi = loan_amount / duration
    emi = math.ceil(emi)
    return emi
```

In the docstring above, we've provided some additional information that the `duration` and `rate` are measured in months. You might even consider naming the arguments `duration_months` and `rate_monthly`, to avoid any confusion whatsoever. Can you think of some other ways to improve the function?

In [ ]:
```
help(loan_emi)
```

```
Help on function loan_emi in module __main__:

loan_emi(amount, duration, rate, down_payment=0)
    Calculates the equal montly installment (EMI) for a loan.

    Arguments:
        amount - Total amount to be spent (loan + down payment)
        duration - Duration of the loan (in months)
        rate - Rate of interest (monthly)
        down_payment (optional) - Optional intial payment (deducted from amount)
```

Exercise - Data Analysis for Vacation Planning

You're planning a vacation, and you need to decide which city you want to visit. You have shortlisted four cities and identified the return flight cost, daily hotel cost, and weekly car rental cost. While renting a car, you need to pay for entire weeks, even if you return the car sooner.

City	Return Flight (<math xmlns="http://www.w3.org/1998/Math/MathML"><semantics><mrow><mi mathvariant="normal">‘</mi><mo stretchy="false">)</mo><mi mathvariant="normal">H</mi><mi>H</mi><mi>o</mi><mi>t</mi><mi>e</mi><mi>l</mi><mi>p</mi><mi>e</mi><mi>r</mi><mi>d</mi><mi>a</mi><mi>y</mi><mo stretchy="false">(</mo><mi mathvariant="normal">’</mi></mrow><annotation encoding="application/x-tex">)	Hotel per day (</annotation></semantics></math>‘)HHotelperday(’)	Weekly Car Rental ($)
Paris	200	20	200
London	250	30	120
Dubai	370	15	80
Mumbai	450	10	70

Answer the following questions using the data above:

1. If you're planning a 1-week long trip, which city should you visit to spend the least amount of money?
2. How does the answer to the previous question change if you change the trip's duration to four days, ten days or two weeks?
3. If your total budget for the trip is `$1000`, which city should you visit to maximize the duration of your trip? Which city should you visit if you want to minimize the duration?
4. How does the answer to the previous question change if your budget is <math xmlns="http://www.w3.org/1998/Math/MathML"><semantics><mrow><mn>600</mn><mi mathvariant="normal">’</mi><mo separator="true">,</mo><mi mathvariant="normal">‘</mi></mrow><annotation encoding="application/x-tex">600, </annotation></semantics></math>600’,‘2000, or `$1500`?

Hint: To answer these questions, it will help to define a function `cost_of_trip` with relevant inputs like flight cost, hotel rate, car rental rate, and duration of the trip. You may find the `math.ceil` function useful for calculating the total cost of car rental.

In [ ]:
```
# Use these cells to answer the question - build the function step-by-step
```

In [ ]:

In [ ]:

In [ ]:

Summary and Further Reading

With this, we complete our discussion of functions in Python. We've covered the following topics in this tutorial:

- Creating and using functions
- Functions with one or more arguments
- Local variables and scope
- Returning values using `return`
- Using default arguments to make a function flexible
- Using named arguments while invoking a function
- Importing modules and using library functions
- Reusing and improving functions to handle new use cases
- Handling exceptions with `try-except`
- Documenting functions using docstrings

This tutorial on functions in Python is by no means exhaustive. Here are a few more topics to learn about:

- Functions with an arbitrary number of arguments using (`*args` and `**kwargs`)
- Defining functions inside functions (and closures)
- A function that invokes itself (recursion)
- Functions that accept other functions as arguments or return other functions
- Functions that enhance other functions (decorators)

Following are some resources to learn about more functions in Python:

- Python Tutorial at W3Schools: https://www.w3schools.com/python/
- Practical Python Programming: https://dabeaz-course.github.io/practical-python/Notes/Contents.html
- Python official documentation: https://docs.python.org/3/tutorial/index.html

You are ready to move on to the next tutorial: "Reading from and writing to files using Python".

Questions for Revision

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is a function?
2. What are the benefits of using functions?
3. What are some built-in functions in Python?
4. How do you define a function in Python? Give an example.
5. What is the body of a function?
6. When are the statements in the body of a function executed?
7. What is meant by calling or invoking a function? Give an example.
8. What are function arguments? How are they useful?
9. How do you store the result of a function in a variable?
10. What is the purpose of the `return` keyword in Python?

11. Can you return multiple values from a function?
12. Can a `return` statement be used inside an `if` block or a `for` loop?
13. Can the `return` keyword be used outside a function?
14. What is scope in a programming region?
15. How do you define a variable inside a function?
16. What are local & global variables?
17. Can you access the variables defined inside a function outside its body? Why or why not?
18. What do you mean by the statement "a function defines a scope within Python"?
19. Do for and while loops define a scope, like functions?
20. Do if-else blocks define a scope, like functions?
21. What are optional function arguments & default values? Give an example.
22. Why should the required arguments appear before the optional arguments in a function definition?
23. How do you invoke a function with named arguments? Illustrate with an example.
24. Can you split a function invocation into multiple lines?
25. Write a function that takes a number and rounds it up to the nearest integer.
26. What are modules in Python?
27. What is a Python library?
28. What is the Python Standard Library?
29. Where can you learn about the modules and functions available in the Python standard library?
30. How do you install a third-party library?
31. What is a module namespace? How is it useful?
32. What problems would you run into if Python modules did not provide namespaces?
33. How do you import a module?
34. How do you use a function from an imported module? Illustrate with an example.
35. Can you invoke a function inside the body of another function? Give an example.
36. What is the single responsibility principle, and how does it apply while writing functions?
37. What some characteristics of well-written functions?
38. Can you use if statements or while loops within a function? Illustrate with an example.
39. What are exceptions in Python? When do they occur?
40. How are exceptions different from syntax errors?
41. What are the different types of in-built exceptions in Python? Where can you learn about them?
42. How do you prevent the termination of a program due to an exception?
43. What is the purpose of the `try-except` statements in Python?
44. What is the syntax of the `try-except` statements? Give an example.
45. What happens if an exception occurs inside a `try` block?
46. How do you handle two different types of exceptions using `except`? Can you have multiple `except` blocks under a single `try` block?
47. How do you create an `except` block to handle any type of exception?
48. Illustrate the usage of `try-except` inside a function with an example.
49. What is a docstring? Why is it useful?
50. How do you display the docstring for a function?
51. What are *args and *kwargs? How are they useful? Give an example.
52. Can you define functions inside functions?
53. What is function closure in Python? How is it useful? Give an example.
54. What is recursion? Illustrate with an example.
55. Can functions accept other functions as arguments? Illustrate with an example.
56. Can functions return other functions as results? Illustrate with an example.
57. What are decorators? How are they useful?
58. Implement a function decorator which prints the arguments and result of wrapped functions.
59. What are some in-built decorators in Python?
60. What are some popular Python libraries?