Functions

1. Introduction

2. Advance usages of Functions

3. Storing your Functions in Modules

Introductions

The best way to develop and maintain a large program is to con struct it from smaller pieces. This technique is called ***divide and conquer***.

The best way to develop and maintain a large program is to con struct it from smaller pieces. This technique is called ***divide and conquer***.

In computer programming, ***abstraction*** refers to the practice of hiding the complexity of an algorithm's sub-steps within a ***function***. Once a function is constructed, it can be treated as a simple expression with defined inputs and outputs, allowing developers to use it without needing to understand its internal details.

The best way to develop and maintain a large program is to con struct it from smaller pieces. This technique is called ***divide and conquer***.

In computer programming, ***abstraction*** refers to the practice of hiding the complexity of an algorithm's sub-steps within a ***function***. Once a function is constructed, it can be treated as a simple expression with defined inputs and outputs, allowing developers to use it without needing to understand its internal details.

We have already seen operations like `print()`, `str()` and `len()`, which involve parentheses wrapped around their arguments. These are examples of Python's built-in functions. Programming language allows us to use a name for a series of operations that should be performed on the given parameters.

The appearance of a function in an expression or statement is known as a ***function call***, or sometimes calling a function.

The appearance of a function in an expression or statement is known as a ***function call***, or sometimes calling a function.

- It allows you to execute a block of codes from various locations in your program by calling the function, rather than duplicating the code.
- It also makes programs easier to modify. When you change a function's code, all calls to the function execute the updated version.

The appearance of a function in an expression or statement is known as a ***function call***, or sometimes calling a function.

- It allows you to execute a block of codes from various locations in your program by calling the function, rather than duplicating the code.
- It also makes programs easier to modify. When you change a function's code, all calls to the function execute the updated version.

A function is a block of organized code that is used to perform a task. They provide better modularity and reusability!

In [2]:
```
display_quiz(path+"func1.json", max_width=800)
```

What is a function in Python?

- A mathematical expression that calculates a value.
- Any sequence of statements.
- A statement of the form x = 5 + 4.
- A named sequence of statements.

`def` Statements with Parameters

When you call the `print()` or `len()` function, you pass them values, called ***arguments***, by typing them between the parentheses. You can also define your own functions that accept arguments.

When you call the `print()` or `len()` function, you pass them values, called ***arguments***, by typing them between the parentheses. You can also define your own functions that accept arguments.

In [3]:
```
def hello(name):
    print('Hello,', name)

hello('Alice')
hello('Bob')
```

```
Hello, Alice
Hello, Bob
```

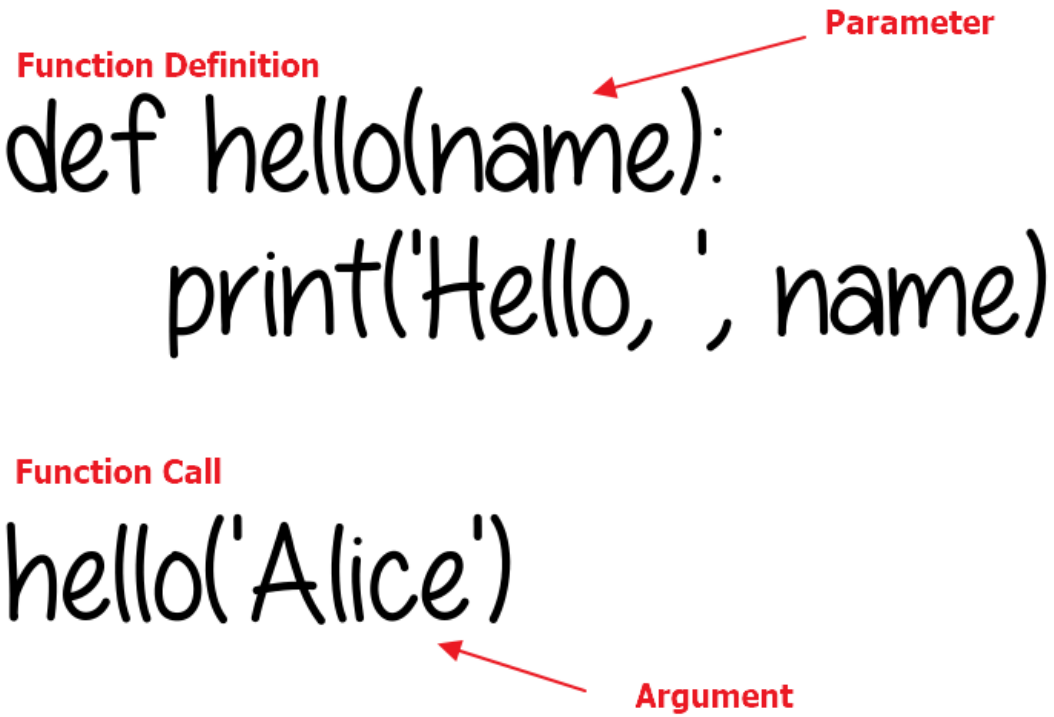
When you call the `print()` or `len()` function, you pass them values, called ***arguments***, by typing them between the parentheses. You can also define your own functions that accept arguments.

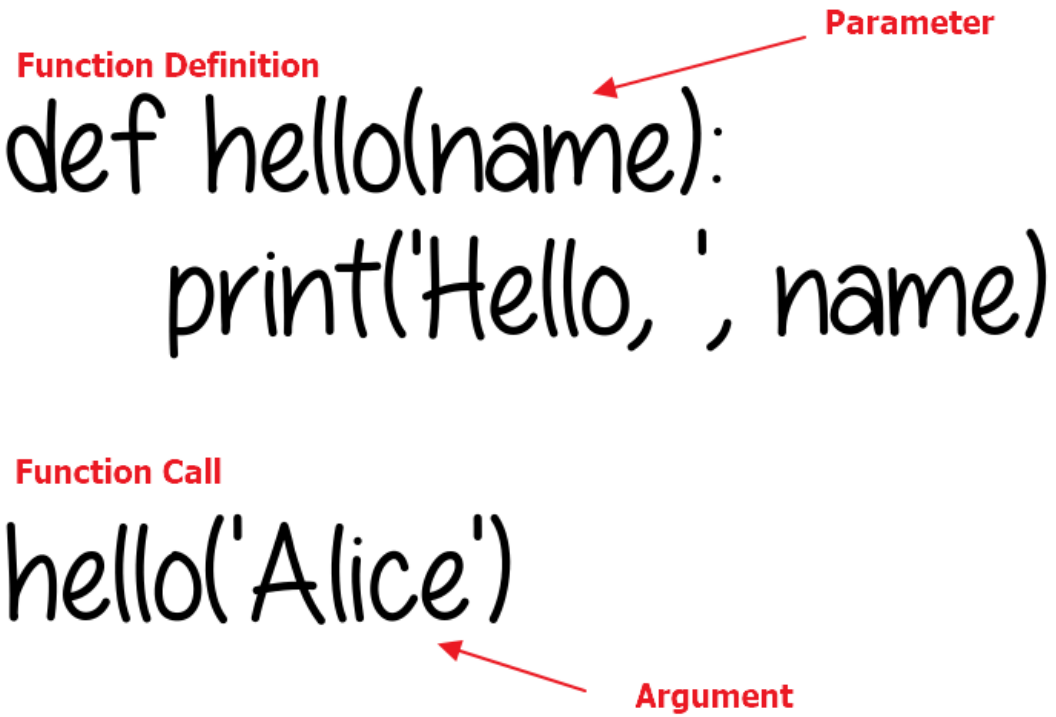
In [3]:
```
def hello(name):
    print('Hello,', name)

hello('Alice')
hello('Bob')
```

```
Hello, Alice
Hello, Bob
```

The `def` statement defines the `hello()` function. Any indented lines that follow `def hello():` make up the function's body. The `hello('Alice')` line calls the function. This function call is also known as passing the string value 'Alice' to the function.


Parameter
Function Definition
def hello(name):
print('Hello, ', name)
Function Call
hello('Alice')
Argument



You can view the execution of this program at https://autbor.com/hellofunc2/. The definition of the `hello()` function in this program has a ***parameter*** called `name`. When a function is called with arguments, the arguments are stored in the parameters.

One thing to note about parameters is that **the value stored in a parameter is forgotten when the function returns**. For example, if you added `print(name)` after `hello('Bob')` in the previous program, the program would give you a `NameError` because there is no variable named `name`.

One thing to note about parameters is that **the value stored in a parameter is forgotten when the function returns**. For example, if you added `print(name)` after `hello('Bob')` in the previous program, the program would give you a `NameError` because there is no variable named `name`.

In [4]:
```
print(name)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_37544\2122694447.py in <module>
----> 1 print(name)

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

In [5]:
```
display_quiz(path+"func2.json", max_width=800)
```

How many lines will be output by executing this code?

```
def hello():
    print("Hello")
    print("NAYAU")
```

3

2

0

Positional Arguments

When you call a function, Python must match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values matched up this way are called ***positional arguments***.

When you call a function, Python must match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values matched up this way are called ***positional arguments***.

In [6]:
```
def describe_pet(animal_type, pet_name):
    """
    Display information about a pet.
    we can write multiple lines here!
    """
    print("\nI have a", animal_type, ".")
    print("My", animal_type + "'s name is", pet_name.title() + ".")

describe_pet('Pokemon', 'Harry')
```

```
I have a Pokemon .
My Pokemon's name is Harry.
```

Note that the text on the second line is a comment called a ***docstring (multi-line comments introduced in Chapter 1)***, which describes what the function does.

Note that the text on the second line is a comment called a ***docstring (multi-line comments introduced in Chapter 1)***, which describes what the function does.

When Python generates documentation for the functions in your programs, it looks for a string immediately after the function's definition. These strings are usually enclosed in triple quotes, which lets you write multiple lines. If you use the `help()` function, it will also be printed out as well as the function name and parameters.

Note that the text on the second line is a comment called a ***docstring (multi-line comments introduced in Chapter 1)***, which describes what the function does.

When Python generates documentation for the functions in your programs, it looks for a string immediately after the function's definition. These strings are usually enclosed in triple quotes, which lets you write multiple lines. If you use the `help()` function, it will also be printed out as well as the function name and parameters.

In [7]:
```
help(describe_pet)
```

```
Help on function describe_pet in module __main__:

describe_pet(animal_type, pet_name)
    Display information about a pet.
    we can write multiple lines here!
```

In [8]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.st
dout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

In [8]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.st
dout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

In [9]:
```
print("8", "9", sep="*")
```

```
8*9
```

Return Values and return Statements

When you call the `len()` function and pass it an argument such as 'Hello', the function call evaluates to the integer value. The value that a function call evaluates to is called the ***return value*** of the function.

When you call the `len()` function and pass it an argument such as 'Hello', the function call evaluates to the integer value. The value that a function call evaluates to is called the ***return value*** of the function.

When creating a function using the `def` statement, you can specify what the return value should be with a ***`return` statement***. A `return` statement consists of the following:

When you call the `len()` function and pass it an argument such as 'Hello', the function call evaluates to the integer value. The value that a function call evaluates to is called the ***return value*** of the function.

When creating a function using the `def` statement, you can specify what the return value should be with a ***`return` statement***. A `return` statement consists of the following:

- The `return` keyword
- The value or expression that the function should return

When an expression is used with a `return` statement, the return value is what this expression evaluates to.

For example, the following program defines a function that returns a different string depending on the number passed as an argument.

For example, the following program defines a function that returns a different string depending on the number passed as an argument.

In [10]:
```
import random

def getAnswer(answerNumber):
    if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidedly so'
    elif answerNumber == 3:
        return 'Yes'
    elif answerNumber == 4:
        return 'Reply hazy try again'
    elif answerNumber == 5:
        return 'Ask again later'
    elif answerNumber == 6:
        return 'Concentrate and ask again'

r = random.randint(1, 6)
fortune = getAnswer(r)
print(fortune)
```

```
Ask again later
```

You can view the execution of this program at https://autbor.com/magic8ball/.

You can view the execution of this program at https://autbor.com/magic8ball/.

When this program starts, Python first imports the `random` module. Then the `getAnswer()` function is defined. Because the function is being defined (and not called), the execution skips over the code in it. Next, the `random.randint()` function is called with two arguments: 1 and 6.

You can view the execution of this program at https://autbor.com/magic8ball/.

When this program starts, Python first imports the `random` module. Then the `getAnswer()` function is defined. Because the function is being defined (and not called), the execution skips over the code in it. Next, the `random.randint()` function is called with two arguments: 1 and 6.

The program execution returns to the line at the bottom of the program that was originally called `getAnswer()`. The returned string is assigned to a variable named `fortune`, which then gets passed to a `print()` call and is printed to the screen. The functions that return values are sometimes called ***fruitful functions***.

In [12]:
```
display_quiz(path+"func3.json", max_width=800)
```

What is wrong with the following function definition:

```
def add_em(x, y, z):
    return x+y+z
    print("the answer is", x+y+z)
```

- You should never use a print statement in a function definition.
- You must calculate the value of x+y+z before you return it.
- A function cannot return a number.
- You should not have any statements in a function after the return statement. Once the function gets to the return statement it will immediately stop executing the function.

The `None` Value

In Python, there is a value called `None`, which represents the absence of a value. The `None` value is the only value of the `NoneType` data type. This can be helpful when you need to store something that won't be confused for a real value in a variable.

In Python, there is a value called `None`, which represents the absence of a value. The `None` value is the only value of the `NoneType` data type. This can be helpful when you need to store something that won't be confused for a real value in a variable.

One place where `None` is used is as the return value of `print()`. The `print()` function displays text on the screen, but it doesn't need to return anything! But since all function calls need to evaluate to a return value, `print()` returns `None`. A function does not return a value is called a ***void function***.

In [13]:
```
spam = print('Hello!')
print(spam)
type(spam)
```

```
Hello!
None
```

Out[13]:
```
NoneType
```

In [13]:
```
spam = print('Hello!')
print(spam)
type(spam)
```

```
Hello!
None
```

Out[13]:
```
NoneType
```

Behind the scenes, Python adds return `None` in the end of any function definition with no `return` statement. Also, if you use a `return` statement without a value (that is, just the `return` keyword by itself), then `None` is returned.

In [14]:
```
display_quiz(path+"func4.json", max_width=800)
```

What is wrong with the following function definition:

- The value None
- The string 'x+y+z'
- The value of x+y+z

Keyword Arguments

A ***keyword argument*** is a name-value pair you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion.

A ***keyword argument*** is a name-value pair you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion.

In [15]:
```
describe_pet(animal_type='Pokemon', pet_name='Harry')
```

```
I have a Pokemon .
My Pokemon's name is Harry.
```

A ***keyword argument*** is a name-value pair you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion.

In [15]:
```
describe_pet(animal_type='Pokemon', pet_name='Harry')
```

```
I have a Pokemon .
My Pokemon's name is Harry.
```

The function `describe_pet()` hasn't changed. But when we call the function, we explicitly tell Python which parameter each argument should be matched with. When Python reads the function call, it knows to assign the argument 'Pokemon' to the parameter `animal_type` and the argument 'Harry' to `pet_name`.

Default parameter values

When writing a function, you can define a ***default parameters***. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value. For example, if you notice that most of the calls to `describe_pet()` are being used to describe dogs, you can set the default value of `animal_type` to 'dog':

When writing a function, you can define a ***default parameters***. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value. For example, if you notice that most of the calls to `describe_pet()` are being used to describe dogs, you can set the default value of `animal_type` to 'dog':

In [16]:
```
def describe_pet(pet_name, animal_type='dog'):
    """
    Display information about a pet.
    Here we have default value for the animal type
    """
    print("\nI have a " + animal_type + ".")
    print("My" + animal_type +"'s name is " + pet_name.title() + ".")

describe_pet('willie')
```

```
I have a dog.
Mydog's name is Willie.
```

Note that the order of the parameters in the function definition had to be changed. **Because the default value makes it unnecessary to specify a type of animal as an argument**, the only argument left in the function call is the pet's name.

Note that the order of the parameters in the function definition had to be changed. **Because the default value makes it unnecessary to specify a type of animal as an argument**, the only argument left in the function call is the pet's name.

Python still interprets this as a positional argument, so if the function is called with just a pet's name, that argument will match up with the first parameter listed in the function's definition.

When you use default values, any parameter with a default value needs to be listed after all the parameters that don’t have default values. This allows Python to continue interpreting positional arguments correctly. Otherwise error occurs.

When you use default values, any parameter with a default value needs to be listed after all the parameters that don't have default values. This allows Python to continue interpreting positional arguments correctly. Otherwise error occurs.

In [17]:
```
def describe_pet(animal_type='dog', pet_name):
    """
    Display information about a pet.
    Here we have default value for the animal type
    """
    print("\nI have a " + animal_type + ".")
    print("My" + animal_type +"'s name is " + pet_name.title() + ".")

describe_pet('willie')
```

```
  File "C:\Users\adm\AppData\Local\Temp\ipykernel_37544\574269134.py",
line 1
    def describe_pet(animal_type='dog', pet_name):
                                                  ^
SyntaxError: non-default argument follows default argument
```

Advance usage

Passing an Arbitrary Number of Arguments (Optional)

Sometimes you won’t know how many arguments a function needs to accept ahead of time. Fortunately, Python allows a function to collect arbitrary arguments from the calling statement.

Sometimes you won't know how many arguments a function needs to accept ahead of time. Fortunately, Python allows a function to collect arbitrary arguments from the calling statement.

For example, consider a function that builds a pizza. It needs to accept a number of toppings, but you can't know ahead of time how many toppings a person will want. The function in the following example has one parameter, `*toppings`, but this parameter collects as many arguments as the calling line provides:

Sometimes you won't know how many arguments a function needs to accept ahead of time. Fortunately, Python allows a function to collect arbitrary arguments from the calling statement.

For example, consider a function that builds a pizza. It needs to accept a number of toppings, but you can't know ahead of time how many toppings a person will want. The function in the following example has one parameter, `*toppings`, but this parameter collects as many arguments as the calling line provides:

In [18]:
```
def make_pizza(size, *toppings):
    """Print the list of toppings that have been requested."""
    print("The size of the pizza is", size, "inch with the following toppings
    print(toppings)

make_pizza(6, 'pepperoni')
make_pizza(8, 'mushrooms', 'green peppers', 'extra cheese')
```

```
The size of the pizza is 6 inch with the following toppings:
('pepperoni',)
The size of the pizza is 8 inch with the following toppings:
('mushrooms', 'green peppers', 'extra cheese')
```

In the function definition, Python assigns the first value it receives to the parameter `size`. All other values that come after are stored in the `tuple` (which we will discuss in later chapters) with the name `toppings`. The function calls include an argument for the size first, followed by as many toppings as needed.

In the function definition, Python assigns the first value it receives to the parameter `size`. All other values that come after are stored in the `tuple` (which we will discuss in later chapters) with the name `toppings`. The function calls include an argument for the size first, followed by as many toppings as needed.

You'll often see the generic ***parameter name*** `*args`, which collects arbitrary positional arguments like this.

Using Arbitrary Keyword Arguments (Optional)

Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides.

Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides.

One example involves building user profiles: you know you'll get information about a user, but you're not sure what kind of information you'll receive.

Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides.

One example involves building user profiles: you know you'll get information about a user, but you're not sure what kind of information you'll receive.

In [19]:
```
def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know about a user."""
    user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')
print(user_profile)
```

```
{'location': 'princeton', 'field': 'physics', 'first_name': 'albert',
'last_name': 'einstein'}
```

The definition of `build_profile()` expects a first and last name, and then it allows the user to pass in as many name-value pairs (Keyword arguments) as they want.

The definition of `build_profile()` expects a first and last name, and then it allows the user to pass in as many name-value pairs (Keyword arguments) as they want.

The double asterisks before the parameter `**user_info` cause Python to create a `dictionary` (Which we will discuss in later chapters) called `user_info` containing all the extra name-value pairs the function receives.

The definition of `build_profile()` expects a first and last name, and then it allows the user to pass in as many name-value pairs (Keyword arguments) as they want.

The double asterisks before the parameter `**user_info` cause Python to create a `dictionary` (Which we will discuss in later chapters) called `user_info` containing all the extra name-value pairs the function receives.

In the body of `build_profile()`, we add the first and last names to the user_info dictionary because we'll always receive these two pieces of information from the user, and they haven't been placed into the dictionary yet. Then we return the `user_info` dictionary to the function call line.

The definition of `build_profile()` expects a first and last name, and then it allows the user to pass in as many name-value pairs (Keyword arguments) as they want.

The double asterisks before the parameter `**user_info` cause Python to create a `dictionary` (Which we will discuss in later chapters) called `user_info` containing all the extra name-value pairs the function receives.

In the body of `build_profile()`, we add the first and last names to the user_info dictionary because we'll always receive these two pieces of information from the user, and they haven't been placed into the dictionary yet. Then we return the `user_info` dictionary to the function call line.

You'll often see the parameter name `**kwargs` used to collect nonspecific keyword arguments. The `**kwargs` must be the rightmost paramter.

In [20]:
```
display_quiz(path+"order.json", max_width=800)
```

Which of the following statement about the function is correct?

```
def func(a, b=2, c, *args, **kwargs):
    return a+b
```

- It will raise a SyntaxError because *args must come before default parameters.
- It will raise a SyntaxError because **kwargs must come before *args.
- It will not raise a SyntaxError.
- It will raise a SyntaxError because non-default parameters cannot follow default parameters.

Exercise 1: Please write a function implementing the ”guess the number” game. The function accepts two arguments for the maximum number of tries and the maximum number. The function returns a boolean value indicating whether the player guessed the number correctly or not. If the player doesn’t guess the number correctly after the maximum number of tries, the function returns False; otherwise, it should return True.

In [ ]:

```python
import random

def guess_number(max_tries, max_number=10):
    """
    Function that allows the player to guess a number between 1 and max_numbe
    If the player can guess the correct number within max_tries times, return
    Otherwise, return False
    """
    # Generate a random number between 1 and max_number
    number = _____

    # Allow the player to guess up to max_tries times
    for i in range(max_tries):
        # Prompt the player to guess the number
        guess = int(input("Guess the number (between 1 and "+ str(max_number)

        # Check if the guess is correct
        if ______:
            print("Congratulations, you guessed the number!")
            ______
        elif ______:
            print("The number is higher than your guess.")
        else:
            print("The number is lower than your guess.")

    # If the player couldn't guess the number in max_tries tries, reveal the
    print("Sorry, you didn't guess the number. The number was " + str(number)
    _______
```

In [ ]:
```python
# Call the function to start the game with a maximum of 5 tries
game_result = guess_number(5)

# Print the result of the game
if game_result:
    print("You won!")
else:
    print("You lost!")
```

Local and Global Scope

Parameters and variables assigned in a called function are said to exist in that function's ***local scope***. Variables assigned outside all functions are said to exist in the ***global scope***.

Parameters and variables assigned in a called function are said to exist in that function's ***local scope***. Variables assigned outside all functions are said to exist in the ***global scope.***

A variable in a local scope is called a ***local variable***, while a variable in the global scope is called a ***global variable***. A variable must be one or the other; it cannot be both local and global.

Think of a scope as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten.

Think of a scope as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten.

There is only one global scope, and it is created when your program begins. A local scope is created whenever a function is called. Any variables assigned in the function exist within the function’s local scope. When the function returns, the local scope is destroyed, and these variables are forgotten.

Think of a scope as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten.

There is only one global scope, and it is created when your program begins. A local scope is created whenever a function is called. Any variables assigned in the function exist within the function’s local scope. When the function returns, the local scope is destroyed, and these variables are forgotten.

The next time you call the function, the local variables will not remember the values stored in them from the last time it was called.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when you run it:

Consider this program, which will cause an error when you run it:

In [21]:
```
def spam():
    eggs = 31337

spam()
print(eggs)
```

```
---------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_37544\4207286030.py in <module>
      3
      4 spam()
----> 5 print(eggs)

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

Consider this program, which will cause an error when you run it:

In [21]:
```
def spam():
    eggs = 31337

spam()
print(eggs)
```

```
---------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_37544\4207286030.py in <module>
      3
      4 spam()
----> 5 print(eggs)

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

The error happens because the `eggs` variable exists only in the local scope created when `spam()` is called. Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs.

In [22]:
```
display_quiz(path+"local.json", max_width=800)
```

What would be the result of running the following code?

10	4
Code will give an error because there are two different y values.	6

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

In [23]:
```
def spam():
    eggs = 99
    bacon()
    print(eggs)

def bacon():
    ham = 101
    eggs = 0

spam()
```

```
99
```

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

In [23]:
```
def spam():
    eggs = 99
    bacon()
    print(eggs)

def bacon():
    ham = 101
    eggs = 0

spam()
```

```
99
```

You can view the execution of this program at https://autbor.com/otherlocalscopes/.

Global Variables Can Be Read from a Local Scope

In [24]:
```
def spam():
    print(eggs)

eggs = 42
spam()
print(eggs)
```

```
42
42
```

In [24]:
```
def spam():
    print(eggs)

eggs = 42
spam()
print(eggs)
```

```
42
42
```

You can view the execution of this program at https://autbor.com/readglobal/. Since there is no parameter named eggs or any code that assigns `eggs` a value in the `spam()` function, when `eggs` is used in `spam()`, Python considers it a reference to the global variable `eggs`. This is why 42 is printed when the previous program is run.

In [25]:
```
def spam():
    eggs = 'spam local'
    print(eggs)    # prints 'spam local'

def bacon():
    eggs = 'bacon local'
    print(eggs)    # prints 'bacon local'
    spam()
    print(eggs)    # prints 'bacon local'

eggs = 'global'
bacon()
print(eggs)        # prints 'global'
```

```
bacon local
spam local
bacon local
global
```

If you want to modify the global variable, use the `global` keywords.

If you want to modify the global variable, use the `global` keywords.

In [26]:
```
def spam():
    global eggs    # If you want to modify the global eggs use global keyword
    eggs = 'spam local'
    print(eggs)    # prints 'spam local'

eggs = 'global'
spam()
print(eggs)
```

```
spam local
spam local
```

If you want to modify the global variable, use the `global` keywords.

In [26]:
```
def spam():
    global eggs    # If you want to modify the global eggs use global keyword
    eggs = 'spam local'
    print(eggs)    # prints 'spam local'

eggs = 'global'
spam()
print(eggs)
```

```
spam local
spam local
```

You can visulaize the execution here.

In [27]:
```
display_quiz(path+"global.json", max_width=800)
```

What would be the result of running the following code?

9	1
10	Error, local variable 'x' is referenced before assignment.

Storing Your Functions in Modules

One advantage of functions is the way they separate blocks of code from your main program. When you use descriptive names for your functions, your programs become much easier to follow.

One advantage of functions is the way they separate blocks of code from your main program. When you use descriptive names for your functions, your programs become much easier to follow.

You can go a step further by storing your functions in a separate file called a ***module*** and then importing that module into your main program. An `import` statement tells Python to make the code in a module available in the currently running program file.

Importing a module

To start importing functions, we first need to create a module. **A module is a file ending in `.py` that contains the code you want to import into your program**. Let's make a module that contains the function `make_pizza()`.

To start importing functions, we first need to create a module. **A module is a file ending in `.py` that contains the code you want to import into your program**. Let's make a module that contains the function `make_pizza()`.

In [28]:
```
%load_ext autoreload
%autoreload 2
```

To start importing functions, we first need to create a module. **A module is a file ending in `.py` that contains the code you want to import into your program**. Let's make a module that contains the function `make_pizza()`.

In [28]:
```
%load_ext autoreload
%autoreload 2
```

In [29]:
```
%%writefile pizza.py
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print("\nMaking a "+ str(size) + "-inch pizza with the following toppings
    print(toppings)
```

```
Overwriting pizza.py
```

In [30]:
```
import pizza

pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
('pepperoni',)

Making a 12-inch pizza with the following toppings:
('mushrooms', 'green peppers', 'extra cheese')
```

In [30]:
```
import pizza

pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
('pepperoni',)

Making a 12-inch pizza with the following toppings:
('mushrooms', 'green peppers', 'extra cheese')
```

When Python reads this file, the line import pizza tells Python to open the file `pizza.py` and copy all the functions from it into this program. You don't actually see code being copied between files because Python copies the code behind the scenes, just before the program runs.

In [30]:
```
import pizza

pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
('pepperoni',)

Making a 12-inch pizza with the following toppings:
('mushrooms', 'green peppers', 'extra cheese')
```

When Python reads this file, the line import pizza tells Python to open the file `pizza.py` and copy all the functions from it into this program. You don't actually see code being copied between files because Python copies the code behind the scenes, just before the program runs.

To call a function from an imported module, enter the name of the module you imported, pizza, followed by the name of the function, `make_pizza()`, separated by a dot.

Importing Specific Functions using `from`

You can also import a specific function from a module.

You can also import a specific function from a module.

In [31]:
```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
('pepperoni',)

Making a 12-inch pizza with the following toppings:
('mushrooms', 'green peppers', 'extra cheese')
```

You can also import a specific function from a module.

In [31]:
```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
('pepperoni',)

Making a 12-inch pizza with the following toppings:
('mushrooms', 'green peppers', 'extra cheese')
```

With this syntax, you don't need to use the dot notation when you call a function.

Importing All Functions in a Module

You can tell Python to import every function in a module by using the asterisk (*) operator:

You can tell Python to import every function in a module by using the asterisk (*) operator:

In [32]:
```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
('pepperoni',)

Making a 12-inch pizza with the following toppings:
('mushrooms', 'green peppers', 'extra cheese')
```

You can tell Python to import every function in a module by using the asterisk (*) operator:

In [32]:
```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
('pepperoni',)

Making a 12-inch pizza with the following toppings:
('mushrooms', 'green peppers', 'extra cheese')
```

The asterisk in the `import` statement tells Python to copy every function from the module pizza into this program file. Because every function is imported, you can call each function by name without using the dot notation.

Using `as` to Give a Function an Alias

If the name of a function you're importing might conflict with an existing name in your program, or if the function name is long, you can use a short, unique alias — an alternate name similar to a nickname for the function.

If the name of a function you're importing might conflict with an existing name in your program, or if the function name is long, you can use a short, unique alias — an alternate name similar to a nickname for the function.

In [33]:
```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
('pepperoni',)

Making a 12-inch pizza with the following toppings:
('mushrooms', 'green peppers', 'extra cheese')
```

Using `as` to Give a Module an Alias

You can also provide an alias for a module name. Giving a module a short alias, like p for pizza, allows you to call the module's functions more quickly.

You can also provide an alias for a module name. Giving a module a short alias, like p for pizza, allows you to call the module's functions more quickly.

In [34]:
```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
('pepperoni',)

Making a 12-inch pizza with the following toppings:
('mushrooms', 'green peppers', 'extra cheese')
```

Exercise 2: In this word game, the player is in a land full of dragons. Some dragons are friendly. Other dragons are hungry and eat anyone who enters their cave. The player approaches two caves, one with a friendly and the other with a hungry dragon, but doesn't know which dragon is in which cave. The player must choose between the two. Please completet the design of game by calling the function from the provided `game` module.

In [ ]:

```
%%writefile word_game.py
import random
import time
_____________

playAgain = 'yes'

while playAgain == 'yes':
    # Display the information of game using the displayIntro() in game module
    _____________
    # Read the user input and return the cave number by calling the function
    caveNumber = _________________
    # Check whether the cave is safe or not by calling the checkCave() in gam
    ______(_____)

    print('Do you want to play again? (yes or no)')
    playAgain = input()
```

In [ ]:
```
%%writefile word_game.py
import random
import time
_____________

playAgain = 'yes'

while playAgain == 'yes':
    # Display the information of game using the displayIntro() in game module
    _____________
    # Read the user input and return the cave number by calling the function
    caveNumber = _________________
    # Check whether the cave is safe or not by calling the checkCave() in gam
    ______(_____)

    print('Do you want to play again? (yes or no)')
    playAgain = input()
```

In [ ]:
```
!python word_game.py
```

In [35]:
```
from jupytercards import display_flashcards
fpath= "flashcards/"
display_flashcards(fpath + 'ch3.json')
```

divide and conquer

Next

>