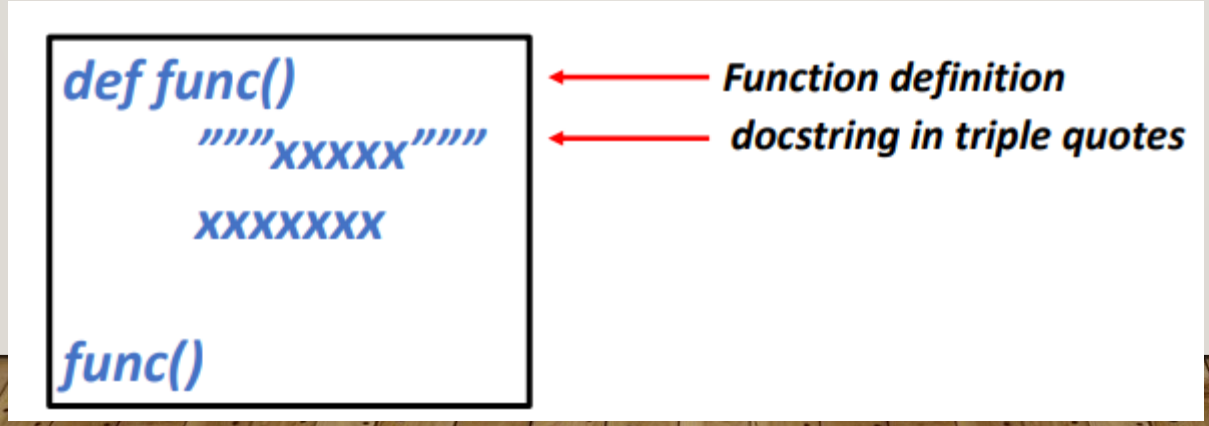
PYTHON

FUNCTION

NOR HAMIZAH MISWAN

FUNCTIONS

- Blocks of code that are designed to do one specific job
- This block of code only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- Using functions makes our programs easier to write, read, test and fix



DEFINING A FUNCTION

- In Python, a function is defined using the def keyword.
- To call a function, use the function name followed by parenthesis.

Try:

```
def my_function():
  print("Hello from a function")

my_function()
```

ARGUMENTS

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.
- The following example has a function with one argument (name). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Try:

```
def my_function(name):
        print(name + “! Hello from a function")
my_function(“Sara")
my_function(“Jerry")
```

ARGUMENTS

- *Arguments* are often shortened to *args* in Python documentations
- Parameters or Arguments?
  - The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function
- From a function's perspective:
  - A parameter is the variable listed inside the parentheses in the function definition.
  - An argument is the value that is sent to the function when it is called.

PASSING ARGUMENTS

- A function may need multiple arguments.
- The arguments can be passed t the functions in a number of ways:
  - Positional arguments
  - Keyword arguments
  - Default values

PASSING ARGUMENTS

- Positional arguments:
  - ➢ Order of the positional arguments matters (need to be in the same order the parameters were written).

Try:

```
def nameAge(name, age):
    print("Hi, I am " + name)
    print("My age is " + str(age))
nameAge("Suraj", 27)
nameAge(27, "Suraj")
```

PASSING ARGUMENTS

- Keyword arguments:
  - Each argument consists of variable name and a value (name-value pair).
  - Keyword arguments free you from having to worry about correctly ordering your arguments in the function call, and they clarify the role of each value in the function call.

Try:

```
def nameAge(name, age):
    print("Hi, I am " + name)
    print("My age is " + str(age))
nameAge(name="Suraj", age=27)
nameAge(age=27, name="Suraj")
```

PASSING ARGUMENTS

- Default values:
  - Order of the parameters in the function definition had to be changed.
  - Parameter with a default value needs to be listed after all the parameters that do not have default values.

Try:

```
def nameAge(name, age=27):
    print("Hi, I am " + name)
    print("My age is " + str(age))
nameAge(name="Suraj")
nameAge("Hassan")
nameAge("Hassan",30)
```

RETURN VALUES

- Process some data and then return a value or set of values.
- When we call a function that returns a value, we need to provide a variable where the return value can be stored.

Try:

```
def get_formatted_name(first_name, last_name):
        """Return a full name, neatly formatted."""
        full_name = first_name + ' ' + last_name
        return full_name.title()
musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

Try:

```
def square_value(num):
    """This function returns the square
    value of the entered number"""
    return num**2
print(square_value(2))
print(square_value(-4))
```

MAKING AN ARGUMENT OPTIONAL

- Sometimes it makes sense to make an argument optional so that people using the function can choose to provide extra information only if they want to.
- You can use default values to make an argument optional.

MAKING AN ARGUMENT OPTIONAL

Try:

```
def get_formatted_name(first_name, last_name, middle_name=''):
        """Return a full name, neatly formatted."""
        full_name = first_name + ' ' + middle_name + ' ' + last_name
        return full_name.title()

musician3 = get_formatted_name('hassan', 'ali')
print(musician3)

musician4 = get_formatted_name('hassan', 'ali', 'mohd')
print(musician4)
```

RETURNING A DICTIONARY

- function can return any kind of value you need it to, including more complicated data structures like lists and dictionaries.

Try:

```
def build_person(first_name, last_name):
    """Return a dictionary of information about a person."""
    person = {'first': first_name, 'last': last_name}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

PASSING A LIST

- It's often find it useful to pass a list to a function, whether it's a list of names, numbers, or more complex objects, such as dictionaries.
- When you pass a list to a function, the function gets direct access to the contents of the list.

Try:

```
def my_function(food):
  for x in food:
    print(x)
fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

PASSING AN ARBITRARY NUMBER OF ARGUMENTS

- Sometimes we won't know ahead of time how many arguments a function needs to accept.
- Python allows a function to collect an arbitrary number of arguments from the calling statement.

Try:

```
def make_pizza(*toppings):
    """Print the list of toppings that have been requested."""
    print(toppings)
 make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

PASSING AN ARBITRARY NUMBER OF ARGUMENTS

- If we want a function to accept several different kinds of arguments, the parameter that accepts an arbitrary number of arguments must be placed last in the function definition.
- Python matches positional and keyword arguments first and then collects any remaining arguments in the final parameter.

Try:

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print("\nMaking a " + str(size) + "-inch pizza with the following toppings:")
    for topping in toppings:
        print("- " + topping)
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

PASSING AN ARBITRARY NUMBER OF ARGUMENTS

- Sometimes we want to accept an arbitrary number of arguments, but we won't know ahead of time what kind of information will be passed to the function.
- In this case, we 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.

PASSING AN ARBITRARY NUMBER OF ARGUMENTS

Try:

```
def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know about a user."""
    profile = {}
    profile['first_name'] = first
    profile['last_name'] = last
    for key, value in user_info.items():
        profile[key] = value
    return profile
user_profile = build_profile('albert', 'einstein', location='princeton', field='physics')
print(user_profile)
```

STORING FUNCTIONS IN MODULE

- Store function in a separate file called module.
- A module: a file ending in .py that contains the code you want to import into our program.
- Storing functions in a separate file allows to hide the details of the program's code and focus on its higher-level logic and allows to reuse functions in many different programs.
- Several ways to import a module:
  - ➢ importing an entire module.
  - ➢ importing specific functions.
  - ➢ importing all functions in a module.

STORING FUNCTIONS IN MODULE

- To import an entire module:
  - importing specific module using import module_name.
  - To call a function from an imported module, enter the name of the module you imported, followed by the name of the function, separated by a dot.

    module_name.function_name()

Try:

```
import module

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

STORING FUNCTIONS IN MODULE

- To import specific functions:
  - the general syntax for this approach:

```
from module_name import function_name
from module_name import function_0, function_1, function_2
```

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

Try:

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

STORING FUNCTIONS IN MODULE

- If the name of a function imported might conflict with an existing name in the program or if the function name is long, we can use a short, unique *alias* (as). How?

```
from module_name import function_name as fn
```

  - Then, we can simply write fn() instead.

Try:

```
from module import make_pizza as mp

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

STORING FUNCTIONS IN MODULE

- We can also provide an alias for a module name. How?

```
import module_name as mn
```

  - The module_name is given the alias mn in the import statement, but all of the module's functions retain their original names.

Try:

```
import module as m


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

STORING FUNCTIONS IN MODULE

- To import all functions in a module.
  - ➢by using the asterisk (*) operator. How?

    from module_name import *
  - ➢The asterisk in the import statement tells Python to copy every function from the module into this program file.
  - ➢Because every function is imported, we can call each function by name without using the dot notation. Try:

```
from module import *
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

PYTHON

CLASSES

NOR HAMIZAH MISWAN

CLASSES

- Combine functions and data into one package
- Represent real-world things and situations
- Create objects based on these classes
- __init__() is a special method in Python that runs automatically whenever we create a new instance
- *self* parameter is required in the method definition → a reference point to the instance

self PARAMETER

- Any variable prefixed with *self* is available to every method in the class.
- *self.name = name* takes the value stored in the parameter name and stores it in the variable name.
- Variables that are accessible through instances like this are called attributes.

Try:

```
class Person:
    """Create a class named Person to assign values for name and age"""
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

MAKING AN INSTANCE FROM A CLASS

- Think of a class as a set of instructions for how to make an instance.
- The class is a set of instructions that tells Python how to make individual instances representing specific class.
- We can create as many instances from a class as we need.
- To access the attributes of an instance, you use dot notation. How?

instance_name.name

Try:

```
p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

MAKING AN INSTANCE FROM A CLASS

- After we create an instance from the class, we can use dot notation to call any method (function) defined in the class. How?

instance_name.method()

Try:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

WORKING WITH CLASSESS AND INSTANCES

- Lets write a new class representing a car:

```
class Car():
    """A simple attempt to represent a car."""
    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()
```

```
my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
```

SETTING A DEFAULT VALUE FOR AN ATTRIBUTE

- Every attribute in a class needs an initial value, even if that value is 0 or an empty string.
- In some cases, such as when setting a default value, it makes sense to specify this initial value in the body of the __init__() method; If we do this for an attribute, we don't have to include a parameter for that attribute.

SETTING A DEFAULT VALUE FOR AN ATTRIBUTE

Try:

```
class Car():
    """A simple attempt to represent a car."""
    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()
```

```
def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + "
miles on it.")

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

Add default value

MODIFYING ATTRIBUTE VALUES

- You can change an attribute's value in three ways:
  - change the value directly through an instance,
  - set the value through a method, or
  - increment the value (add a certain amount to it) through a method

MODIFYING ATTRIBUTE VALUES

- To change the value directly through an instance,
  - ➢modify attribute value directly

Add:

```
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

MODIFYING ATTRIBUTE VALUES

- Modify by set the value through a method,
  - Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.

Add new method to the previous class:

```
def update_odometer(self, mileage):
    """Set the odometer reading to the given value."""
    self.odometer_reading = mileage
```

```
my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

MODIFYING ATTRIBUTE VALUES

- Sometimes you'll want to increment an attribute's value by a certain amount rather than set an entirely new value
  - ➢ increment the value (add a certain amount to it) through a method

Add new method to the previous class:

```
def increment_odometer(self, miles):
    """Add the given amount to the odometer reading."""
    self.odometer_reading += miles
```

```
my_used_car = Car('subaru', 'outback', 2013)
print(my_used_car.get_descriptive_name())

my_used_car.update_odometer(23500)
my_used_car.read_odometer()

my_used_car.increment_odometer(100)
my_used_car.read_odometer()
```

INHERITANCE

- We don't have to start from scratch when writing a class •
- Use inheritance → parent class and child class
- *Super* comes from a convention of calling the parent class a *superclass* and the child class a *subclass*.
- *Super() function* needs two arguments: a reference to the child class and the self object.

INHERITANCE

- As an example, let's model an electric car. An electric car is just a specific kind of car, so we can base our new ElectricCar class on the Car class we wrote earlier.
- Then we'll only have to write code for the attributes and behavior specific to electric cars.

Add new class to the previous class:

```
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""
    def __init__(self, make, model, year):
        """Initialize attributes of the parent class."""
        super().__init__(make, model, year)
```

```
my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
```

INHERITANCE

- Once you have a child class that inherits from a parent class, you can add any new attributes and methods necessary to differentiate the child class from the parent class.

INHERITANCE

Add an attribute that's specific to electric cars:

```
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery_size = 70

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print("This car has a " + str(self.battery_size) + "-kWh battery.")
```

```
my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

IMPORTING CLASS AS MODULE

- Python lets you store classes in modules and then import the classes you need into your main program.
- Importing a single class:

  from module_name import class_name
- Importing multiple class from a module:

  from module_name import class_name1, class_name2
- Importing entire module:

  import module_name

  module_name.method()
- Importing all classes from a module:

  from module_name import *

IMPORTING CLASS AS MODULE

Save class car and electric car as module. Then try:

```
from car import Car

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

```
from car import ElectricCar

my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

IMPORTING CLASS AS MODULE

Save class car and electric car as module. Then try:

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla', 'roadster', 2016)
print(my_tesla.get_descriptive_name())
```

IMPORTING CLASS AS MODULE

Save class car and electric car as module. Then try:

```
#import entire module
import car

my_beetle = car.Car('volkswagen', 'beetle', 2016)
print(my_beetle.get_descriptive_name())

my_tesla = car.ElectricCar('tesla', 'roadster', 2016)
print(my_tesla.get_descriptive_name())
```