Intermediate Python

Programming for Professionals

Classes

Object Oriented Programming

OBJECT ORIENTED

Until now, you have essentially done procedural programming. With the introduction to classes, you will begin writing object-oriented programs with Python. Object-oriented programming is not just about classes. Indeed you can write procedural programs using classes. Procedural versus object-oriented programming is akin to straight-drive versus automatic vehicles. Both get you to the desired destination – but the methodology and rewards are entirely different.

Object oriented programming is modeling a domain specific problem and implementing a solution. Concepts such as inheritance, encapsulation, and abstraction, are important in this modeling paradigm.

Teaching the concepts of object-oriented programming is beyond the scope of this course. In this module, the focus is primarily on syntax.

NUTS AND BOLTS

In a real-world problem domain, you have nouns, verbs, and adjectives. For example, if the domain is payroll, the nouns are employees, paycheck, payroll period, company, and so on. The verb phrases are doing payroll, calculating taxes, printing paychecks, printing paystubs, etc.

The adjectives are exempt, monthly (payroll), etc. In objected oriented programming, nouns are classes, verbs are methods, and adjectives are properties.

Another important concept is inheritance, which implies an "is a" kind of relationship. From a technical perspective, inheritance is for code reuse where a child class can inherit the behavior and properties of a base class, which reduces redundancy. An exempt employee is a kind of Employee. An hourly employee is a kind of Employee. This implies an Employee base class with ExemptEmployee and HourlyEmployee as child classes.

CLASS

Classes are a template for building objects. Conceptually, a class is a construction blueprint, where the objects are houses. You can build zero or more houses from the same blueprint. In Python, you build objects/instances from classes. Objects are instances of classes. Like the real world, objects have behavior and attributes. This is called encapsulation in object-oriented terminology.

Here is the most basic class:

```
class Employee:
  pass
```

SELF

Do you have a sense of yourself? Do you know what belongs to yourself versus someone else? An object in Python has the same knowledge. That is the concept of *self*. In other languages, this concept is embodied by *this* reference or pointer.

Touch your *self*.nose. Is there a danger that you may touch someone else's nose? Hope not! Works for Python objects the same way. Within the class, use the dot syntax to bind self to instance members of an object:

- self.method()
- self.prop

The first parameter of instance methods is *self*, which is passed into the method implicitly. This makes *self* available within methods as a variable.

CONSTRUCTORS

Objects should always be in a known state. Uninitialized data is a common problem. The constructor is a special function (_init_) responsible for creating an instance of an object: creating a known state from the beginning. What do you initialize? You initialized the attributes and properties that belong to the object. In our example, that is the firstname and lastname of an Employee.

Constructors are functions that initialize an object. Constructors are instance methods and the first parameter is *self*.

Here is an example constructor:

```
class Employee:
  def _init_(self):
    pass
```

DESTRUCTORS

Objects call constructors (__init__) to initialize objects. Conversely, destructors (__del __) are called implicitly to release resources associated with an object. Deleting an object (del command) will invoke the destructor.

```
>>>
>>> class XClass:
...     def __del__(self):
...             print("relinguish resources")
...
>>>
>>> obj=XClass()
>>> del obj
relinguish resources
>>>
>>>
```

DATA MEMBERS

Instance data members contain data belonging to an object and defined in the constructor (self.*datamember*). You bind a data member to an object using the dot syntax. Within the class, that is self.*datamember*.

Binding outside the class is accomplished with *objectname.datamember*. All data members should be initialized in the constructor with the method parameters..

```
def_init_(self, _firstname, _lastname):
    self.firstname=_firstname
    self.lastname=_lastname
```

In the preceding code, the underscore for the parameters is a naming convention and not required.

OBJECT INSTANCES

You are now ready to create an instance of a class, an object, and assign to a variable. The object is created in memory and the variable name is the label.

The syntax is:

```
variablename=classname(constructor parameter list)
```

Here is sample code:

```
emplobject=Employee("Donis", "Marshall")
print(emplobject.firstname)
print(emplobject.lastname)
```

After an object is created, refer to members of the instance using the dot syntax.

CLASSES ARE NAMESPACES

You can import classes as a namespace. In this manner, you can create another namespace level.

```
# MyClass.py

class XClass:

      def FuncA():

            print("FuncA")


      def FuncB():

            print("FuncB")


      def FuncC():

            print("FuncC")


# UseMyClass.py

from myclass import XClass


XClass.FuncA()
```

CLASS VS INSTANCE NAMESPACES

A namespace is a mapping from names to objects

Python classes and instances of a class each have their own distinct namespaces represented by the attributes ClassName.__dict__ and object.__dict__ respectively.

The order for resolving names is: Instance namespace first, then class namespace if it does not find the attribute in the instance namespace.

CLASS ATTRIBUTE ASSIGNMENT VIA CLASS

If a class attribute is set using the class, it will override the value for all instances. At the namespace level, this sets Xclass.__dict__['class_var']

```
obj1, obj2 = XClass(), XClass()
XClass.class_var = 200

print('Obj1 -',obj1.class_var)
print('Obj1 -',obj2.class_var)
```

✓ 0.2s

```
Obj1 - 200
Obj1 - 200
```

CLASS ATTRIBUTE ASSIGNMENT VIA INSTANCE

If a class attribute is set using the INSTANCE, it will override the value for ONLY for that instance. At the namespace level, class_var is added to obj1.__dict__.

```
obj1, obj2 = XClass(), XClass()
obj1.class_var = 100

print('Obj1 -',obj1.class_var)
print('Obj1 -',obj2.class_var)
```

✓ 0.3s

```
Obj1 - 100
Obj1 - 200
```

WHEN TO USE CLASS ATTRIBUTES

- Storing constants
- Defining default values
- Tracking data across instances of a class

GETATTR()

Getattr() is used to access the value of an attribute that's bound to an object.

Syntax: getattr(obj, key, def)

Returns: object value if available

```
class ZClass:

    def __init__(self):
        self.data = 'Test'

obj = ZClass()
print('Data attr =',getattr(obj,'data'))
# Default value
print('XData attr =',getattr(obj,'xdata','Not set'))
# No default value
print('ZData attr =',getattr(obj,'zdata'))
```

⊗ 0.5s

```
Data attr = Test
XData attr = Not set

---------------------------------------------------------------
AttributeError                            Traceback (most re
c:\Users\ishea\Documents\Python3Lessons\Intro_To_Python\inte
      7 print('Data attr =',getattr(obj,'data'))
      8 print('XData attr =',getattr(obj,'xdata','Not set'))
----> 9 print('ZData attr =',getattr(obj,'zdata'))

AttributeError: 'ZClass' object has no attribute 'zdata'
```

STATIC AND CLASS METHODS

Classwise methods belong to a class instead of an instance. The first parameter is *cls* for the class method. The *cls* argument is passed implicitly. Within the method, use *cls* to manage classwise variables.

Decorate classwise methods with @classmethod attribute. The best example of a classwise method and variable is counting instances.

Static methods do not belong to either an instance or class. The surrounding class is essentially a namespace.

STATIC AND CLASS METHODS – EXAMPLE CODE

```
class XClass:

    count=0

    @classmethod

    def Func1(cls):

        cls.count=cls.count+1

    @staticmethod

    def Func2(cls):

        cls.count=cls.count+1 # not work

        pass


obj=XClass()

XClass.Func1()

XClass.Func2()
```

METHOD TYPES

```
class MyClass:
    def method(self):
        self.__class__.test=15
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        cls.test+=1
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'

    def staticmethod2():
        return 'static method2 called'
```

```
obj=MyClass()
print(obj.method())
print(MyClass.method(obj))

print(obj.classmethod())
print(MyClass.classmethod())

print(MyClass.test)

print(MyClass.staticmethod())
print(MyClass.staticmethod2())
```

PROPERTIES

What is wrong with this code?

```
class Person:

  def _init_(self):

    self.age=1


bob=Person()

bob.age=45
```

What prevents the age from being set to 1000? Only Moses lived to 1000 years old! Access to the data member is unfettered. For that reason, the data member can be set improperly.

Furthermore, this breaks abstraction.

PROPERTIES - 2

Is this a better solution? Encapsulation binds behavior and data within an object while exposing the public interface and hiding the data. State is changed through behavior. This allows the validation of data.

In the example, the age data member is hidden by naming convention. You must set and get the age via two methods: GetAge and SetAge.

```
class Person:
	def init (self):
		self.__ageunknown=1

	def SetAge(self, _age):
		if(_age < 0 or _age > 120):
			return False
		self.__ageunknown=_age
		return True

	def GetAge(self):
		return self.__ageunknown

bob=Person()
if(bob.SetAge(54)):
	print(bob.GetAge())
```

PROPERTIES - 3

It would be nice to have the best of both worlds: the convenience of data members and the computational capabilities of methods. Properties provide both. There are three methods associated with a property: @property, @*property*.setter, and @*property*.deleter methods. When a value is assigned to the property, the @*property*.setter method is called. When the value is retrieved, the @property method is called. When the property is deleted, the @*property*.deleter method is called.

Sample code for properties is next.

PROPERTIES – EXAMPLE CODE

```
class Person:
        def_init_(self):
                self.__ageunknown=1

        @property
        def age(self):
                return self.__ageunknown

        @age.setter
        def age(self, _age):
                if(_age < 0 or _age > 120):
                        raise Exception("Invalid Age")
                self._ageunknown=_age

        @age.deleter
        def age(self):
                raise TypeError("Can't Delete")
```

```
bob=Person()
try:
    bob.age=54
    print(bob.age)
    del bob.age
except Exception as e:
    print(e)
```

PROPERTIES NOTES

- If the set method fails, throw an exception.
- Keep the property specific methods short – maximum of 5 lines
- Read-only properties only implement the @property method
- Write-only properties implement only the @*property*.setter method.

PROPERTIES / DELETE

```
class P:

    def __init__(self,x):
        self.x = x

    def __del__(self):
        print("destructor called")

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x

    @x.deleter
    def x(self):
        print("deleter called on property")


obj=P(10)
del obj
```

POP QUIZ: PROPERTIES

Give examples of both write- and read-only properties?

5 MINUTES

VALIDATION USING PROPERTIES

When two or more properties have the same validation logic, redundancy occurs.

```
class XClass:

    def __init__(self,firstname,lastname):
        self.__firstname, self.__lastname = firstname, lastname


    @property
    def firstname(self):
        return self.__firstname

    @firstname.setter
    def firstname(self, value):
        if not isinstance(value,str):
            raise ValueError("firstname must be a string")

        if len(value) == 0:
            raise ValueError("firstname cannot be empty")

        self.__firstname = value

    @property
    def lastname(self):
        return self.__lastname

    @lastname.setter
    def lastname(self, value):
        if not isinstance(value,str):
            raise ValueError("lastname must be a string")

        if len(value) == 0:
            raise ValueError("lastname cannot be empty")

        self.__lastname = value
```

DESCRIPTORS

A descriptor is an object of a class that implements one of the methods specified in the descriptor protocol. Descriptors are used to allow objects to customize:

- Attribute lookup
- Attribute storage
- Attribute deletion

DESCRIPTOR PROTOCOL

The descriptor protocol consists of three methods:

__get__ - gets an attribute

__set__ - sets an attribute value

__delete__ deletes an attribute

__set_name__ (optional) can be used to set an attribute on an instance of a class

DESCRIPTOR TYPES

There are two types of descriptors:

- Data descriptor: implements __set__ and/or __delete__
- Non-data descriptor implements __get__ only

The descriptor type specifies the property lookup resolution.

DESCRIPTOR USING CLASS METHIDS

In this example RequiredField class implements method from the descriptor protocol and is a data descriptor.

The descriptor is used by creating it's instance as an attribute in a user class. It can be reused across many different classes and attributes.

```
class RequiredField:

    def __set_name__(self, owner, name):
        print(f'__set_name__ was called with owner:{owner} and name:{name}')
        self.property_name = name

    def __get__(self,instance,owner):
        print(f'__get__ was called with instance:{instance} and owner:{owner}')
        if instance is None:
            return self

        return instance.__dict__[self.property_name] or None

    def __set__(self,instance,value):
        print(f'__set__ was called with instance:{instance} and value:{value}')

        if not isinstance(value,str):
            raise ValueError(f"{self.property_name} must be a string")

        if len(value) == 0:
            raise ValueError(f"{self.property_name} cannot be empty")

        instance.__dict__[self.property_name] = value

class Person:

    firstname = RequiredField()
    lastname = RequiredField()
```

INHERITANCE

Inheritance implies code reuse. The child class is inheriting code from the base class, including methods and data members. Members are inherited as full members of the child class and accessible via the dot syntax.

This is the syntax for inheritance:

```
class child(base):
    pass
```

Multiple inheritance is supported in Python. But this is different from multiple inheritance in other languages, such as C++. In Python, multiple inheritance is mostly evaluated from left and right.

Here is the syntax:

```
class child(base¹, base², base³, baseⁿ):
    pass
```

OVERRIDING

Child class can implement an inherited method. The child class implementation will override the base class implementation. The child version of the method must have the same signature as the parent method.

When searching for a method, overriding always searches up the hierarchy from the derived to base class.

OVERRIDING CONSTRUCTORS

Constructors are inherited but can be overridden in the child class. You can call the base class constructor in the overridden method with super(). This allows the invocation of both constructors when needed. The same approach works for other overridden methods.

```
class Parent:
    def __init__(self):
        print("Parent c'tor")

class Child(Parent):
    def __init__(self):
        super().__init__()
        print("Child c'tor")

obj=Child()
```

MULTIPLE INHERITANCE

For multiple inheritance, provide a list of base classes instead of one. Python does support multiple inheritance but differently from C++, which is the most well-known implementation. The Python model is more similar to multi-level single inheritance. This avoids some of the ambiguity problems of the C++ approach. However, the Python model has its own share of potential problems.

The best practice is to avoid multiple inheritance if at all possible.

MULTIPLE INHERITANCE - 2

When inheriting the same method from multiple base classes, Multiple Resolution Order (MRO) defines the search order. The search order is:

1. Current class
2. Immediate base classes (parent) in textual order (left to right)
3. Second level of base classes (grandparents) in textual order

```
class One:
    def FuncA(self):
        print("One.FuncA()")


class Two:
    def FuncA(self):
        print("Two.FuncA()")


class Three:
            pass


class Child(One, Two, Three):
            pass


obj=Child()
obj.FuncA()
```

MULTIPLE INHERITANCE - 2

You can use the class attribute __mro__ to list the Multiple Resolution Order for a particular type.

This is the MRO for the Child class in the example code:

```
(<class '__main__.Child'>, <class
'__main__.One'>, <class '__main__.Two'>,
<class '__main__.Three'>, <class 'object'>)
```

```
class One:

    def FuncA(self):

        print("One.FuncA()")


class Two:

    def FuncA(self):

        print("Two.FuncA()")


class Three:

            pass


class Child(One, Two, Three):

            pass


print(Child.__mro__)
```

MULTIPLE INHERITANCE - 3

Here is the MRO for the Child.

```
<class '__main__.Child’>
      <class '__main__.L1_1’>
            <class '__main__.L2_1’>
                  <class '__main__.L3_1’>
      <class '__main__.L1_2’>
            <class '__main__.L2_2’>
      <class '__main__.L1_3’>,


      <class 'object'>
```

```
class L3_1:
      pass


class L2_1(L3_1):
      pass

class L2_2:
      pass

class L1_1(L2_1):
      pass

class L1_2(L2_2):
      pass

class L1_3:
      pass

class Child(L1_1, L1_2, L1_3):
      pass
```

MULTIPLE INHERITANCE - 4

Here is the MRO for the Child class.

Notice L1_1 and L1_2 inherit from the same class: L2_1.

```
<class '__main__.Child’>
     <class '__main__.L1_1’>
     <class '__main__.L1_2’>
          <class '__main__.L2_1’>
               <class '__main__.L3_1’>
     <class '__main__.L1_3’>


     <class 'object'>)
```

```
class L3_1:
     pass


class L2_1(L3_1):
     pass


class L1_1(L2_1):
     pass


class L1_2(L2_1):
     pass


class L1_3:
     pass


class Child(L1_1, L1_2, L1_3):
     pass
```

MULTIPLE INHERITANCE - 5

In Python 3, C3 Linearization is used for method resolution, which is similar to MRO. C3 / MRO does not resolve all potential ambiguity.

```
Traceback (most recent call last):
  File "C:/Code/Python/Test.py", line 13, in <module>
    class Child(L1_1, L1_2):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases L2_1, L2_2
```

Python Method Resolution Order and C3 linearization algorithm (pilosus.org)

```
class L2_1:
    pass

class L2_2:
    pass

class L1_1(L2_1, L2_2):
    pass

class L1_2(L2_2, L2_1):
    pass

class Child(L1_1, L1_2):
    pass
```

POP QUIZ: WHICH FUNCA IS CALLED?

```
class Grandparent:
    def FuncA(self):
        print("Grandparent.FuncA")


class Parent(Grandparent):
    def FuncA(self):
        print("Parent.FuncA")


class Child(Parent):
    pass


obj=Child()
obj.FuncA()
```

5 MINUTES

LAB 7: SHAPES

30 Minutes

This lab:

- One of the most documented examples of object oriented programming is Shapes
- Create a class for common shapes: Rectangle, Triangle, and Ellipse.

SHAPES

Ellipse, Rectangle, and Triangle are types of Shapes. This forms a "is a kind of" relationship, which implies inheritance.

In this example, Shape would be the base class, which Ellipse, Rectangle, and Triangle as child classes.

Shape has three instance members exposed as get and set "properties". The properties are red, green, and blue (i.e., RGB). Together they set the color of the shape.

Red, green, and blue properties represent integer data members, hidden by convention, and between – 0 and 255.

The constructor of the base class sets the properties and the initial color.

Create three child classes: Ellipse, Rectangle, Triangle. They inherit from Shape. Each has a Draw method instance method that displays the object type plus the color. For example:

```
Ellipse - 12 255 125
```

SHAPES - 2

Each derived type should also have a constructor that delegates to the base class constructor and passes up required parameters.

In main, create a list of three objects via constructors: Rectangle, Triangle, and Circle. You can use any colors – your choice. In a for loop, call Draw on each object.

Decorators

Function Modifiers

DECORATORS

DECORATORS

Decorators are function modifiers. A decorator takes a function as a parameter and wraps another function (modifier) around it. After “doing something”, the decorator typically returns the modifier function.

What is “does something”? That can be anything:

- Modify the parameters
- Perform some sort of calculation
- Validation
- Auditing
- Debugging
- Of course, one option is “do nothing”

DECORATORS - 2

Decorators make “do something” reusable. You can affix the decorator to a target method using the “@” prefix and the decorator name. There are advantages of declarative over imperative programming.

- One of the most important is that declarative programming is more extensible.
- Second, declarative programming is more readable.
- More easily refactored

BACK TO THE EGGS

You have an application where eggs() are an important method. However, there are many types of eggs:

- Fried
- Boiled
- Poached
- Raw

Some of these modifiers are reusable.

- You can fry chicken, fish, vegetables, and potatoes
- You can eat raw fish (sushi), beef (tartare), and vegetables

DECORATOR

Decorators are essentially function wrappers.

```
def register(func):
    print("running registation (%s)"
          %func.__name__)
    return func


@register
def func1():
    prin
```

DECORATOR – NESTED METHOD

Decorators typically use a nested method to encapsulate the behavior of the decorator.

```
def deco(func):
        def inner():  # does the work
            print("test")
            func()
        return inner


@deco
def target():
    print("running target")


target()
```

FRIEND EGGS

Opposite is the code for fried eggs. The *fried* method is a modifier for the egg method.

What are the shortcomings?

- Tight coupling
- Not reusable
- Imperative

```
def egg(number):
    print("You have %d egg(s)"%number, end="")


def fried(number): if not number:
        raise Exception("Number equal 0")
    abs(number)
    egg(number)
    quantity="are" if number > 1 else "is"
    print(", which %s fried."%quantity)


fried(2)
```

THE DECORATOR WAY

This is similar code but done with decorators, which has advantages.

Now @fried represents reusable code that can be applied to steaks, fish, potatoes, and even ice cream.

```
def fried(func):
   def new_function(*args, **kwargs):
      func(*args, **kwargs)
      number=args[0]
      quantity="is/are"
      if(isinstance(number,int)):
             if not number:
                    raise Exception("Number equal 0")
             number=abs(number)
             quantity="are" if number > 1 else "is"
       print(", which %s fried."%quantity)
   return new_function

@fried
def egg(number):
       if(isinstance(number, int)):
             print("You have %d egg(s)"%abs(number), end="")
       else:
             print("You have eggs", end="")
egg(12)
```

POP QUIZ: YOUR GUESS

How might the following code be useful?

```
def guess(func):  # NOP
      def new_function(*args, **kwargs):
            pass
      return new_function


@guess  #temporary deprecation
def testfunc():
      print("test func")
```

5 MINUTES

DECORATOR – STACKED

Decorators can be stacked and are called in order.

For example:

decorator1(decorator2(func))

```
def decorator1(func):
    print("decorator1")

def decorator2(func):
    print("decorator2")


@decorator1  # decorate def decorator2(func)
@decorator2  # decorate def Func1()
def Func1():
    print("func")
```

DECORATOR – PARAMETERS

Decorators can be called with parameters. The syntax is slightly different than previously.

```
def decoratorp(a, b):
    def decoratori(func):
        print(a,b)
        return func
    return decoratori

@decoratorp(4,5)
def Func():
    pass
```

DECORATOR CLASS

Instead of a decorator method, you can create a decorator class. In the decorator class, you must have a two argument constructor: self, fptr. A function pointer to the decorated method is passed as the second argument.

When the decorated function is called, the __call__ method is called from the decorator class. In __call__, you typically wrap a function call to the decorated method with some behavior.

```
class decorator(object):

    def __init__(self, func):

        self.func = func


    def __call__(self, *args):

        self.func(*args)

        print(f"{args[0] + args[1]}")


@decorator

def func(x,y):   # resolve to decObj

    print(f"{x} + {y} = ",end="")


if __name__ == '__main__':

    func(1,2)
```

DECORATORS

Lab 8

LAB: INSTRUMENT

This lab:

Create a decorator that instruments any method.

DECORATOR FOR INSTRUMENTATION

1. Import time module
2. Create a decorator called *instrument*. It takes one parameter called *func*.
3. Inside of the decorator, create another method called *modifier*.
4. In the modifier:
    1. Display the original function name: "*functionname* starting"
    2. Save the current time into a variable.
    3. Call original function.
    4. Save the current time into a separate variable.
    5. Calculate the elapsed time in seconds
    6. Display the function name: "*functionname* ended (*time* seconds)"
5. Return the modifier method.

DECORATOR FOR INSTRUMENTATION - 2

5. Create a method called testmethod
6. Inside of testmethod, sleep for 5 seconds.
7. Add the instrument decorator to the testmethod.
8. Run testmethod. Are the results correct?

Regular Expressions

WHAT IS A REGEX

A Regular Expression (RegEx) is a sequence of characters that define a search pattern.

Regular expressions can be concatenated to form new regular expressions. If X and Y are RegEx, then XY is also a RegEx.

In Python, you can construct and use regular expressions using the re module.

RE MODULE

You can use the re module to create regular expressions.

The example on the right finds any five-letter string starting with an a and ending with an s.

```
import re

pattern = '^a.....d$'
test_string = 'android'
result = re.match(pattern, test_string)

if result:
    print("Pattern matched.")
else:
    print("No match found.")
```

✓ 0.1s

```
Pattern matched.
```

METACHARACTERS

To specify regular expressions, metacharacters are used. In the above example, ^ and $ are metacharacters.

The following is a list of supported metacharacters:

^

$

*

.

+

?

{}

[]

()

\

|

[ ] – SQUARE BRACKETS

Square brackets are used to specify a list of characters to match.

[abc] – matches a string that contains a or b or c

[a-e] – the same as [abcde]

[1-5] – the same as [12345]

[^abc] – matches any string that does not contain a or b or c.

[^0-9] – means any non-digit character

```
pattern = '[abc.]'
text = 'I am a regular expression.'
print(re.findall(pattern,text))
```

✓ 0.4s

```
['a', 'a', 'a', '.']
```

. PERIOD

A period matches any single character (except newline '\n').

. Matches any one character

.. Matches any two characters

... Matches any three characters

```
pattern = '..'
text = 'I am a regular expression.'
num_matches = len(re.findall(pattern,text))
print('Matches:',num_matches)
```

✓ 0.4s

```
Matches: 13
```

^ CARET

The caret is used to test if a string starts with a certain character or character sequence.

^a – starts with a

^ab – starts with ab

```
import re
# Caret
pattern = '^The'
texts = ['The dogs were barking',\
    'There were no dogs in the house',\
        'Yet, she kept on moving forward.']
for text in texts:
    num_matches = len(re.findall(pattern, text))
    print(num_matches,':',text)
```

✓ 0.9s

```
1 : The dogs were barking
1 : There were no dogs in the house
0 : Yet, she kept on moving forward.
```

$ DOLLAR

The dollar sign is used to test if a string end with a certain character or character sequence.

a$ – ends with a

house$ - ends with house

```
import re
# Caret
pattern = 'house$'
texts = ['The dogs were barking',\
    'There were no dogs in the house',\
        'Yet, she kept on moving forward.']
for text in texts:
    num_matches = len(re.findall(pattern, text))
    print(num_matches,':',text)
```

✓ 0.3s

```
0 : The dogs were barking
1 : There were no dogs in the house
0 : Yet, she kept on moving forward.
```

* ASTERISK

The asterisk matches zero or more occurrences of the pattern on it's left.

ma*n – any string with a followed by n

```
import re
# *
pattern = 'd*ing'
texts = ['The rate of flow gradually changed.',\
    'Paddington changed his name to Ted.',\
        'The shading of the text is different.']
for text in texts:
    num_matches = len(re.findall(pattern, text))
    print(num_matches,':',text)
```

✓ 0.4s

```
0 : The rate of flow gradually changed.
1 : Paddington changed his name to Ted.
1 : The shading of the text is different.
```

+ PLUS

The asterisk matches zero or more occurrences of the pattern on it's left.

Ma+n – any string starting with man then followed by n

```
import re
# *
pattern = 'd*ing'
texts = ['The rate of flow gradually changed.',\
    'Paddington changed his name to Ted.',\
        'The shading of the text is different.']
for text in texts:
    num_matches = len(re.findall(pattern, text))
    print(num_matches,':',text)
```

✓ 0.4s

```
0 : The rate of flow gradually changed.
1 : Paddington changed his name to Ted.
1 : The shading of the text is different.
```

? QUESTION MARK

The question mark matches zero or one occurrences of a pattern.

wi?n – starts with *wi* followed by 0 or 1 (any) character and an *"n"*

```
import re

pattern = "wi?n"
values = ['wn','wnn','win','wiiin'\
'wayne','winning','won']

for v in values:
    num_matches = len(re.findall(pattern, v))
    print(num_matches,':',v)
```

✓ 0.7s

```
1 : wn
1 : wnn
1 : win
0 : wiiinwayne
1 : winning
0 : won
```

{} BRACES

Braces can be used to find exactly a specific number of occurrences.

```
import re

pattern = 'a{1,3}' # At least 1 a and at most 3 a's
text = ['abion aaaaand brian','ab road aab caaat']
for t in text:
    num_matches = re.findall(pattern, t)
    print(num_matches,':',t)
```

✓ 0.2s

```
['a', 'aaa', 'aa', 'a'] : abion aaaaand brian
['a', 'a', 'aa', 'aaa'] : ab road aab caaat
```

EITHER OR

The question mark matches zero or one occurrence of a pattern.

- one n or more after ?
- one i followed by n after ?

```
# alternation
pattern = 'hello|ola'

# match any string that contains hello or olla
words = ['"Hello!", yelled the squire.', \
    'King Olaf of Norway','Helios the space company']
for w in words:
    print(w,re.findall(pattern,w.lower()),sep=':\t\t')
```

✓ 0.6s

```
"Hello!", yelled the squire.:           ['hello']
King Olaf of Norway:            ['ola']
Helios the space company:               []
```

() CAPTURE AND GROUP

Parentheses are used to group sub patterns

```
# group
pattern = '(hello|ola), how'

# match any string that contains hello or olla
words = ['Hello, how are you today?',\
    'Hello is a word used for greeting']
for w in words:
    print(w,re.findall(pattern,w.lower()),sep=':\t\t')
```

✓ 0.2s

```
Hello, how are you today?:              ['hello']
Hello is a word used for greeting:              []
```

SPECIAL SEQUENCES

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

RE FUNCTIONS

There are several functions and constants defined in the re module to work with RegEx.

findall()

split()

sub()

subn()

search()

FINDALL

The findall function returns a list of strings containing all matches. If no match is found, an empty list

```
import re

txt = "The impact of COVID-19 on\
    supply chains was not good for business."
x = re.findall("supply", txt)
print(x)
```

✓ 0.3s

```
['supply']
```

SPLIT

The split() function splits the string where there is a match and returns a list of strings

```
import re

string = "The impact of COVID-19 on business."
pattern = '\s+'

result = re.split(pattern, string)
print(result)
```

✓ 0.3s

```
['The', 'impact', 'of', 'COVID-19', 'on', 'business.']
```

SUB()

Sub() replaces all matched occurrences of a pattern in a string with a value specified assigned to the *replace variable.*

```
# Program to remove all whitespaces
import re

# multiline string
phone_number = '673 9374 893'
# matches all whitespace characters
pattern = '\s+'
# empty string
replace = ''
new_phone_number = re.sub(pattern, replace, phone_number)
print(new_phone_number)
```

✓ 0.3s

```
6739374893
```

SEARCH()

The search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

If the search is successful, search() returns a match object; if not, it returns None.

```
import re

string = "COVID-19 lockdowns were a major influence of \
  transition to hybrid work."

# check if 'COVID-19' is at the beginning
match = re.search('\ACOVID-19', string)

if match:
  print("pattern found inside the string")
else:
  print("pattern not found")

print(match)
```

✓ 0.6s

```
pattern found inside the string
<re.Match object; span=(0, 8), match='COVID-19'>
```

THE MATCH OBJECT

The match object is returned after a call to *re.search().* Match exposes the following methods:

- group()
- start()
- end()
- span()

```
string = '73838 847, 3778 3834'

# 4 digit number followed by space
# followed by 3 digit number
pattern = '(\d{4}) (\d{2})'
match = re.search(pattern,string)

# group() returns a part of
# the string where there is a match
print('Group:',match.group(1,2))

# start() returns index of start of matched string
# end() returns index of end of matched string
print('Start:',match.start())
print('End:',match.end())

#span returns a tuple containing the start
# and end of matched string
print('Span:',match.span())
```

ADDITIONAL RESOURCES

https://docs.python.org/3/library/re.html

https://developers.google.com/edu/python/regular-expressions

https://cheatography.com/davechild/cheat-sheets/regular-expressions/

Regular Expressions

Lab 10

LAB: BABY NAMES

45 Minutes

This lab was adopted from Google for Education:

Given a collection of html files with baby names, extract the names, the year and rank numbers.

1. Open babynames in the labs directory
2. Follow the steps in instructions.txt

APPENDIX

Fun Stuff

Tips and Tricks

ITERTOOLS

Itertools package is packed with nifty features. For example, the chain.from_iterable command flattens a sequence.

```
import itertools

a = [[1, 2], [3, 4], [5, 6]]

print(list(itertools.chain.from_iterable(a)))
```

itertools — Functions creating iterators for efficient looping — Python 3.10.7 documentation

POP QUIZ: ALTERNATE TERNARY

The ternary statement is an abbreviated if..else. Here is an example:

```
'true' if True else 'false'
```

There are alternate syntaxes for defining a ternary operation. You just have to be creative. For example:

```
print(("it is odd", "it is even")[a%2==0])
```

5 MINUTES

ENUMERATE

When iterating a sequence, it is often beneficial to extract the next value *and* the corresponding index. The enumerate command does just that!

```
mylist = [1,13,16,15,80]
for i, value in enumerate(mylist):
        print( i, ': ', value)
```

PROFILING

Have you ever had a performance problem? Unable to locate the bottleneck? The profile command (cProfile) can help.

```
python -m cProfile my_script.py
```

```
C:\Code\Python>python -m cProfile example2.py
         8 function calls in 7.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    7.001    7.001 example2.py:1(<module>)
        1    0.000    0.000    7.001    7.001 example2.py:11(main)
        1    0.000    0.000    7.001    7.001 example2.py:3(FuncA)
        1    0.000    0.000    2.000    2.000 example2.py:7(FuncB)
        1    0.000    0.000    7.001    7.001 {built-in method builtins.exec}
        2    7.000    3.500    7.000    3.500 {built-in method time.sleep}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Prof
iler' objects}
```

MEMBERSHIP

You can confirm membership and whether an item is included in a sequence with the if…in… statement.

```
>>> if 4 in [1,2,3,4]:
...     print("membership confirmed")
...
membership confirmed
>>>
```

```
>>> if "a" in {"a":1, "b":2, "c":3}:
...     print("membership confirmed")
...
membership confirmed
>>> _
```

PPRINT

Pretty print provides extra formatting specification when printing data, especially sequences or aggregate data. Import pprint for this feature. Setup pprint by calling PrettyPrinter with format rules. You can then print using the returned object, with the pprint method.

```
>>>
>>> data=list(range(0,20))
>>> print(data)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> p=pprint.PrettyPrinter(indent=4, width=20)
>>> p.pprint(data)
[   0,
    1,
    2,
    3,
    4,
    5,
    6,
    7,
    8,
    9
```

UNPACKING

This is a splat (*) – otherwise known as an asterisk. When passing a parameter with a splat, Python will unpack a sequence as separate parameters.

```
def foo(a, b, c, d, e):
        print(a, b, c, d, e)


mydict = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5}
mylist = [10, 20, 30, 40, 50]
myset=set("Hello!")
mylist2=[10, 20, 30]
mylist3=[40, 50]

foo(*mydict)
foo(**mydict)
foo(*mylist)
foo(*myset)
foo(*mylist2, *mylist3)
```

```
C:\Code\Python>example2.py
a b c d e
1 2 3 4 5
10 20 30 40 50
e o l ! H
10 20 30 40 50

C:\Code\Python>
```

NAMED TUPLE

Named tuples are tuples where each item is associated with a symbolic name. Call the function namedtuple, which is a factory function. Found in the collections module. It creates named attributes for each tuple item.

```
import collections

Point = collections.namedtuple('Point', ['x', 'y'])
p = Point(x=1.0, y=2.0)
print(p.x, p.y)
```

CONSTANT

You can also use named tuples to create the equivalent of a constant in Python.

```
from collections import namedtuple


Constants = namedtuple('Constants', ['pi', 'e'])
constants = Constants(3.14, 2.718)
print(constants.pi)
constants.pi = 3
```

CONSTANT - 2

You can also embed constants, using named tuples, in classes.

```
from collections import namedtuple

class XClass:
        constants = namedtuple('Constants', ['pi', \
                'e'])(3.14, 2.718)

print(XClass.constants.pi)
# XClass.constants.pi = 3
```

ENUMERATION

Unlike other languages, Python does not support enums. You can used the following syntax to create something similar to enumeration however.

```
>>> class Style:
...     Bold, Underline, Italics, Strikethrough=range(4)
...
>>> print(Style.Underline, Style.Italics)
1 2
>>>
```

UNPACKING – VERSION 2

A function may return a sequence that contains a single item. In that circumstance, it might be more convenient to have a scalar value versus a sequence with one element.

The following code unpacks a single element from a sequence with one item.

```
def FuncA():
        return [1]


a,=FuncA()
print(a)
```

Extra Credit

Lab 8

LAB: EXTRA CREDIT

This lab:

Work on extra credit exercise.

Cryptography

Hashing and Encryption

CONCEPTS

CRYPTOGRAPHY

According to the Handbook of Theoretical Computer Science, here is the definition of Cryptography :

Cryptography or cryptology (from Ancient Greek: **κρυπτός**, translit. kryptós "hidden, secret"; and **γράφειν** graphein, "to write", or **-λογία** -logia, "study", respectively[1]) is the practice and study of techniques for secure communication in the presence of third parties called adversaries.

- Cryptography is about protecting sensitive or confidential data with algorithms and protocols designed to hide secret data or detect tampering. Encryption is the process of transforming data to hide a secret.
- Hashing is the process of creating a fixed length result whereupon tampering of the original data is detectable to assure integrity. Man in the middle (MTM) attacks are commonly addressed with hashing algorithms.

HASHING

Hashing is used to detect tampering.

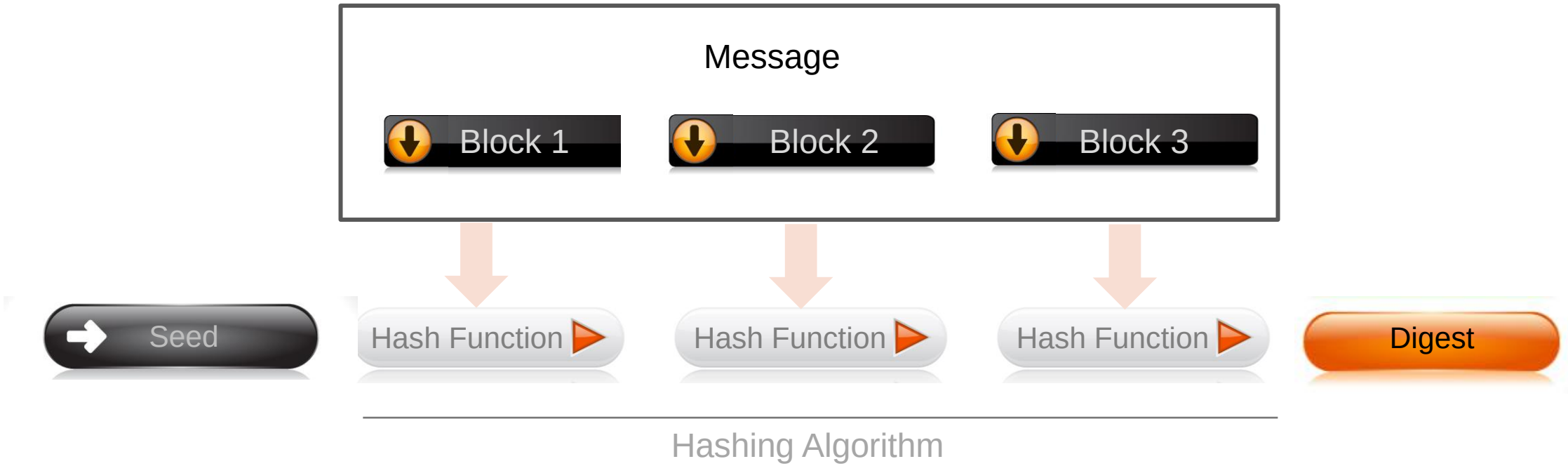
A hash algorithm will convert variable length data into a fixed length encoded result, which is called a digest. Any change to the data, even a single bit, can cause a radically different digest being generated using the hash algorithm.

Here are the attributes of a secure hash algorithm:

- One way – must use brute force to reverse
- Should be fast
- Memory efficient

Hash algorithms do not encrypt data.

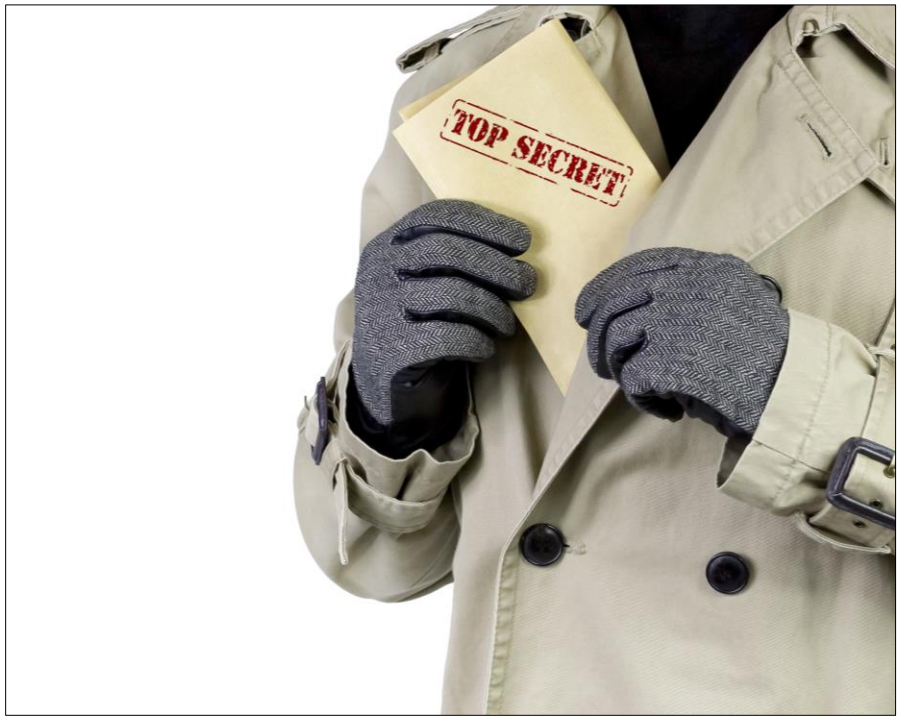
HASHING



ENCRYPTION

Encryption is about keeping secrets confidential. Conversely, decryption discloses the secret.

Python supports various encryption algorithms with different size keys. In general, encryption algorithms with larger keys sizes are more secure. But this is relative concept - as computational capability increases larger keys sizes should be made available to compensate for the additional throughput.



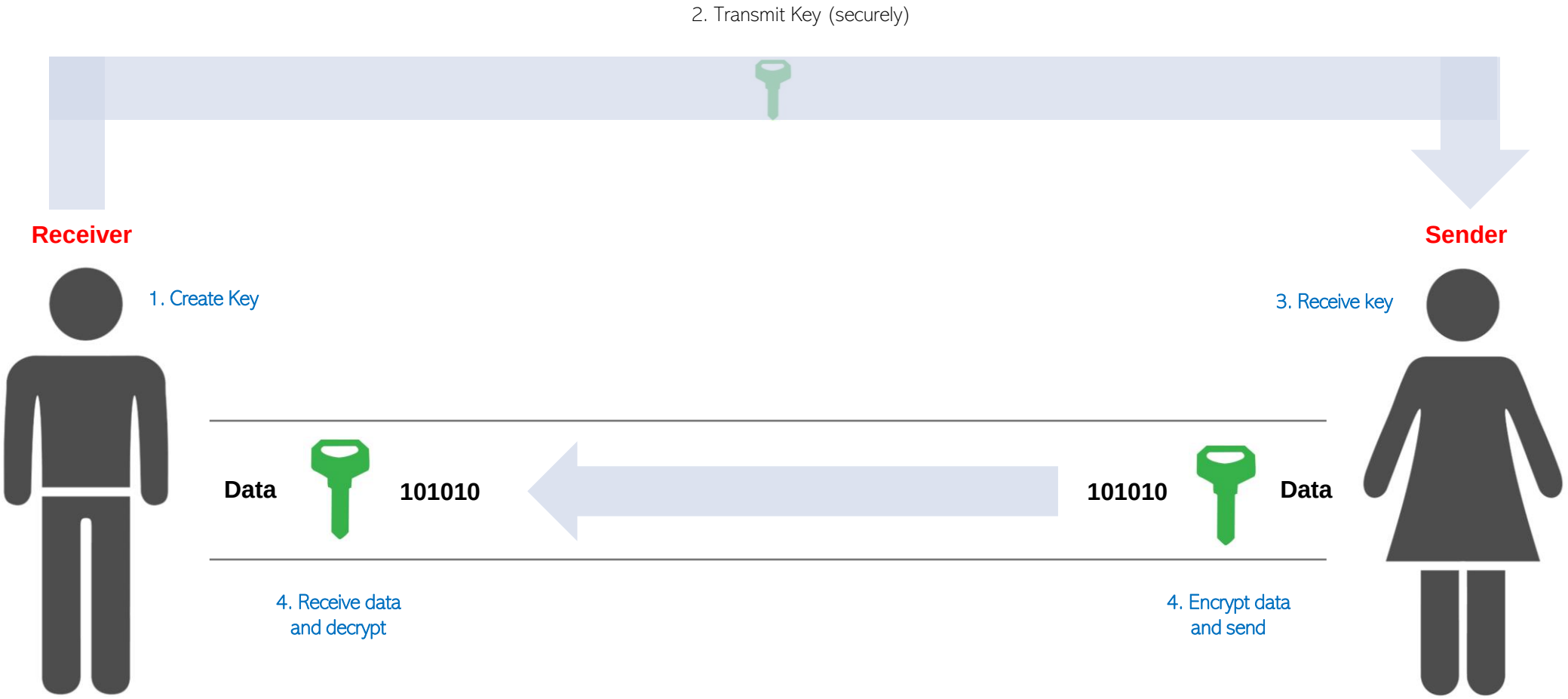
SYMMETRIC VERSUS ASYMMETRIC

A symmetric key is employed to both encrypt (code) and decrypt (decode) data. Conversely, asymmetric keys imply two keys: public key and private key.

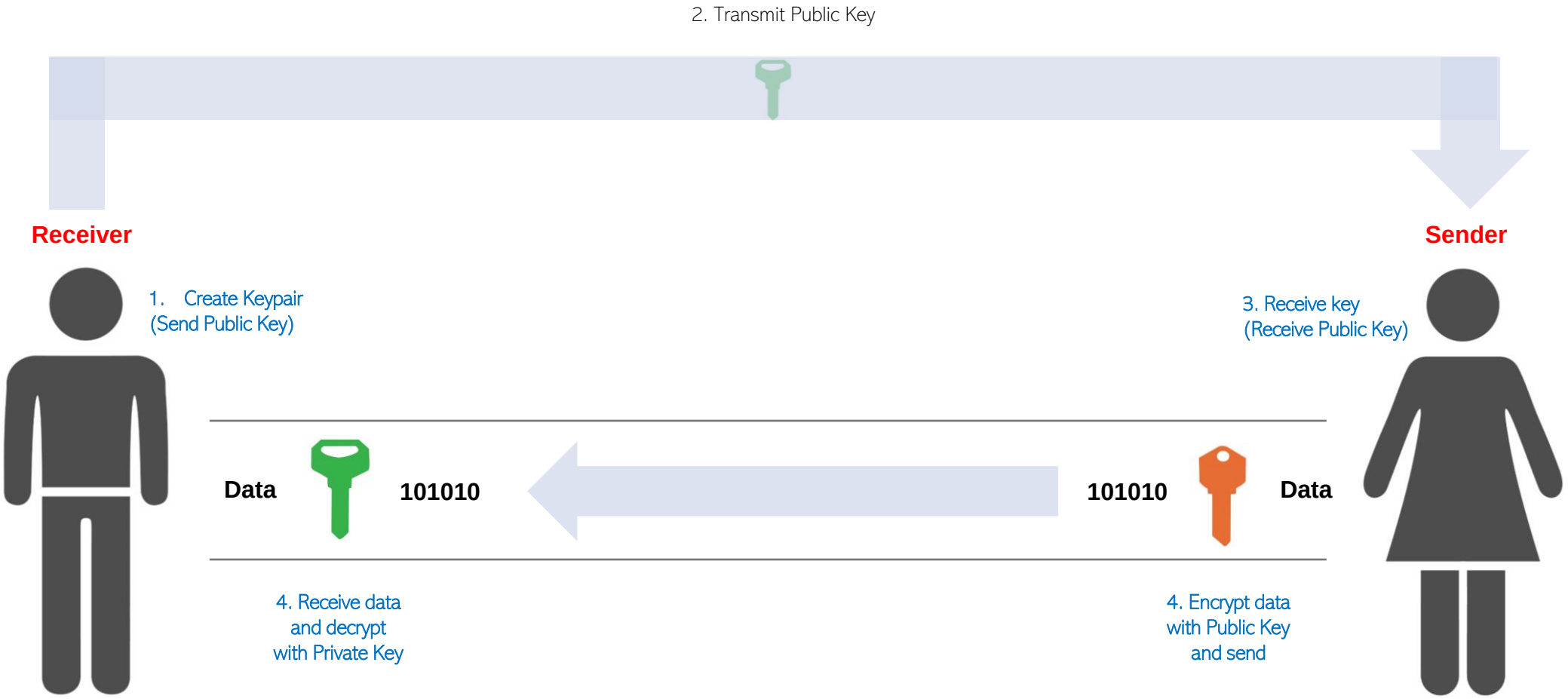
The challenge with symmetric key encryption is both the sender and receiver must agree upon the same key. This needs to be done in a secure manner.

With asymmetric key encryption, the receiver keeps a private key and provides senders a public key. Since the public key is *public*, it does not have to be transmitted securely. There is a relationship between the public and private key, where only the holder of the private key can decrypt data that was encrypted with the public key.

SYMMETRIC - DIAGRAM



ASYMMETRIC - DIAGRAM



IMPLEMENTATION

DATA SIZE

The size of source data is not unlimited. Hashing large amounts of data may not be as secured. The reason is called the ripple effect. The random seed is applied to the first message block. As successive message blocks are encode the effect of the initial seed lessens. If the data is large enough, where the hash function is run hundreds, maybe thousands of times, the effect of the original seed is limited. When this occurs, it is possible for data to have the same hash (collision).

HASHING

Hashing is a one-way algorithm that performs a transformation on data (message) and results in an encoded fixed length result (hash digest). If the data is changed, even minutely, the result can be an entirely different hash. The only attack for a properly generated hash is a brute force attack, which is computationally not practical.

Source data is partitioned into sub blocks of the same size. The size of the digest depends on the hashing algorithm as documented on the next table.

HASHLIB

The HashLib package, contains a variety of hash algorithms implemented in functions.

Here are the SHA2 hash algorithms. Less secure hash algorithms, such as MD5, are also available and useful in certain circumstances.

Class	Function Name	Message Size	Hash Size
SHA256 algorithm	sha256	2^{64}	256
SHA384 algorithm	sha384	2^{384}	384
SHA512 algorithm	sha512	2^{128}	512

HASHING EXAMPLE

Here is example code for creating a hash. In this example, a message (“Hello, world”) is converted into bytes and then hashed. Finally, the hash is displayed.

The sha256 function creates a hash generator: SHA-256 hash object. Call hexdigest function to actually generate the hash as bytes.

```
import hashlib

hash_string = "helllo"

hash_bytes = hash_string.encode()

print("String bytes: ", hash_bytes)

hash_engine = hashlib.sha256(hash_bytes)

digest = hash_engine.hexdigest()

print("Digest bytes: ", digest)
```

HASHING EXAMPLE - 2

```
String bytes:  b'hello'

Digest bytes:
2cf24dba5fb0a30e26e83b2ac5b9e29e1b1
61e5c1fa7425e73043362938b9824

Digest string / base58:
21RyLADfubTbKZWgaC9sqV1iCzR8xypqYHK
ZqazSMZK7jibHB1M8qLx4tYpEJJ8w95xfNt
9eux8i1KbunAc6pNm1
```

```
import hashlib

hash_string = "hello"

hash_bytes = hash_string.encode()

print("String bytes: ", hash_bytes)

hash_engine = hashlib.sha256(hash_bytes)

digest = hash_engine.hexdigest()

print("Digest bytes: ", digest)
```

SYMMETRIC ENCRYPTION

Encryption helps an application protect confidential data. There are two types of encryption: symmetric and asymmetric.

With symmetric, there is one key that must be securely shared between two applications.

SYMMETRIC ENCRYPTION - 2

Encryption is about transformation of a data into oblique data, which keeps secrets confidential. Encryption algorithms use a cipher function to transform each block of data, also know as plaintext, into blocks of encrypted data. For this reason, the encryption function is called a block cipher, while the encrypted data is called the cipher.

Encryption is a two-way process. You can reverse the encryption algorithm (decryption) and restore plain text to the original data.

SYMMETRIC ENCYPTION

Encryption in Python is simpler than most languages. The concepts remain a challenge for many but the implementation is easy.

In this example, the Ferret.generate_key method creates a key, which is used to encrypt and decrypt a message. Most of the details of what is occurring is abstracted.

```
from cryptography.fernet import Fernet


key = Fernet.generate_key()
f = Fernet(key)
edata= f.encrypt(b"secret message.")


val=f.decrypt(edata)
print(val))
```

ASYMMETRIC ENCRYPTION

With asymmetric, a private and public key are generated. Keep the private key and send the public key to anyone sending you confidential data.

The public key is used to encrypt data and create a cipher, while the private key is used to decrypt the cipher and restore the plain text.

If you are sending confidential data, the receiver (not you) creates a public and private key; then forwards the public key. You use public key to encrypt and send the secret.

Transmission of the public key does not have to be secure.

ASYMMETRIC ENCRYPTION - 2

Private and public keys are mathematically related. Each encryption algorithms is responsible for defining that relationship. With asymmetric encryption, there are separate block cipher functions to encrypt (public key) and decrypt (private key) data.

Asymmetric encryption is slower than symmetric encryption. In addition, the maximum size of the data (plain text) is smaller.

POP QUIZ: ASYMMETRIC ENCRYPTION

What is the most common use of the asymmetric encryption and why?

10 MINUTES

Lab 8- Simple Blockchain

EMERGING TECHNOLOGY

25 Minutes

What is blockchain?

Blockchain has ushered in the next generation of web applications.

Despite this, the answer to the preceding question is difficult for many. Blockchain is much more than Bitcoin, which is a cryptocurrency. With recent financial speculation on Bitcoin, interest in cryptocurrency has gone viral. Blockchain is part of the Bitcoin technology stack. However, Blockchain is larger than cryptocurrencies.

WHAT IS BLOCKCHAIN?

Blockchain is a decentralized internet-based application that hosts a distributed ledger. Imagine a single accounting spreadsheet managed simultaneously by multiple Microsoft Excel web servers. The ledger consists of transactions. The transactions can range from financial debits and credits to requests to execute functionality. Transactions can literally be anything depending on the blockchain application.

ATTRIBUTES OF A BLOCKCHAIN

Blockchain is a protocol for a decentralized application hosting a distributed ledger. Blockchains that follow that protocol share similar attributes.

- No central authority
- Immutable
- Secure
- Transparent
- Privacy
- Scalable
- Available

BLOCK

Create a class Block. The block should have these fields:

- hash (string)
- previousHash (string)
- data (string)
- timeStamp (time)
- count (int) – this is a static field
- blockId (int)

FUNCTIONS

The block class has two methods:

- CalculateHash
  - This method has no parameters
  - Combines the fields of the block, except hash, to create a string.
  - Using the steps demonstrated in this module, create a hash from the combined string.
  - Return the hash
- Constructor
  - This is an instance constructor
  - There are three parameters: self, data, previousHash

FUNCTIONS - 2

- Assign the previousHash parameter to the previousHash field of the current block
- Assign the data parameter to the data field of the block current block.
- Set timestamp to be the current time
- Increment the static count
- Update the blockId using the static count
- Set hash with the CalculateHash function

MAIN

Define and call a *main* function.

In main:

- Define blockchain as an empty list
- Append a new block to the blockchain
  - Data: “I’m the genesis block”
  - previousHash=“”
- Append another new block to the blockchain
  - Data: “I’m the second block”
  - Hash from the previous block
- Append a third block to the blockchain
  - Data: “I’m the third block”
  - Hash from the previous block

Iterate the blockchain and display the blockId, hash, and previous hash of every block.