Python 3
Object Oriented syntax

I OBJECT-ORIENTED PROGRAMMING

I.0 What is object oriented programming ?

- The basic idea is that you create *objects* as building blocks for your application.

- If you are building a kitchen, for example, you might need objects like toasters, blenders and can openers. If you are building a large kitchen, you might have many different toasters, but they all have the same characteristics.
  They all belong to the same *class*, in this case a class called `Toaster`.

- Each object has some data associated with it. A toaster might have a certain heat setting and a crumb tray that collects the crumbs that fall off each time it toasts bread. Each toaster has its *own* heat setting and its *own* crumb count, so each `Toaster` object has its own variables to represent these things.

- In object speak, these variables are called *instance variables* or *attributes*. You can use these instead of global variables to represent your data.

- You tell an object to do something using special procedures called *methods* or *attributes*. For example, a `Toaster` object might have a method called `toast` that you use to toast bread, and another method called `clean` that you use to clean out the crumb tray.

- Methods let you define a few strictly limited ways to access the data in a class, which helps you prevent many errors.

- The `constructor` are special methods that are called automatically when an object is created.

- Everything that you need to know about an object is described in its *class definition*.
  The class definition lists the instance variables that hold an object's data and the methods that are used to manipulate the object. It acts like a blueprint for creating objects. Objects themselves are often called *instances* of the class that they belong to.

- The elements of a class are hidden inside its scope, so they do not clutter the global Tcl name space. When you are inside a method or class procedure, you can directly name other methods, class procedures, and the various class variables.

- When you are outside a class, you can only invoke its methods through an object.

- Usually, the methods are the public part of an object, and the variables are kept hidden inside.

- Specific methods and variables may exist within a class: class methods and class variables.

    - The class methods and class variables are shared by all objects in a class.
    - The instance methods and instance variables are per-object.

I.1 Classes and Instances

- A class is a Python object with several characteristics:
    - You can call a class object as if it were a function. The call creates another object, known as an ***instance of the class***, which knows what class it belongs to.
    - A class has arbitrarily named ***attributes*** that you can bind and reference.
    - The values of class attributes can be data objects or function objects.
- Class attributes bound to functions are known as ***methods*** of the class.
- A class can ***inherit*** from other classes, meaning it can delegate to other class objects the lookup of attributes that are not found in the class itself.
- An instance of a class is a Python object with arbitrarily named attributes that you can bind and reference. An instance object implicitly delegates to its class the lookup of attributes not found in the instance itself. The class, in turn, may delegate the lookup to the classes from which it inherits, if any.
- In Python, classes are objects (values), and are handled like other objects. For instance, you can pass a class as an argument in a call to a function.
- The fact that classes are objects in Python is often expressed by saying that classes are first-class objects.

I.2 The class Statement

- The **`class`** statement is the most common way to create a class object.

- `class` is a compound statement with the following syntax:

```
class classname[(base-classes)]:
    statement(s)
```

  `classname` is an identifier.

  `base-classes` is an optional comma-delimited list of class objects. These classes are known as the ***base classes***, ***superclasses***, or ***parents*** of the class being created. The class being created is said to ***inherit from***, ***derive from***, ***extend***, or ***subclass*** its base classes. This class is also known as a direct ***subclass*** or ***descendant*** of its base classes.

  The non-empty sequence of *`statements`* that follows the `class` statement is known as the ***class body***. A class body executes immediately, as part of the class statement's execution.

Note: The subclass relationship between classes is transitive.

- The built-in function **`issubclass(class1, class2)`** returns `True` if *`class1`* subclasses *`class2`*, otherwise it returns `False`.

- In Python 3 classes do always subclass (directly or indirectly) the built-in class **`object`** .

I.3 The class body

- The body of a class is where you normally specify the attributes of the class; these attributes can be data objects (instance or class variables) or function objects (instance or class methods).

I.3.1 Class variables

- You typically specify a ***class variable*** by binding a value to an identifier within the class body.

```
class Date (object):
     nbOfMonths = 12
print(Date.nbOfMonths)              # prints: 12
```

- Class object `Date` now has one class variable named *`nbOfMonths`* bound to the value 12 and *`Date.nbOfMonths`* refers to this variable.

- You can also bind or unbind class variables outside the class body. For example:

```
class Date (object):pass
Date.nbOfMonths = 12
print(Date.nbOfMonths)
```

- Any class variable is implicitly shared by all instances of the class.

- The `class` statement implicitly defines some class variables:
  - **`__name__`** is the class-name identifier string used in the class statement.
  - **`__bases__`** is the tuple of class objects given as the base classes in the class statement.
  - **`__dict__`** is a dictionary object that the class uses to hold all of its other attributes.
  - **`__module__`** is the module name in which the class is defined.

I.3.2 Reference to class attributes

- In statements that are directly in a class's body, references to class variables of the class must use directly the name of the variable:

- However, in statements that are in methods defined in a class body, references to variables of the class must use a **fully qualified name**:

```
class Test:
    x = 5
    y = x + 6 # direct use of x
    def amethod(self):
        print(Test.x)
```


I.3.3 Instance Methods

- A class body may include **`def`** statements. These functions (called ***instance methods*** in this context) are important attributes for class objects.

- A `def` statement in a class body obeys the rules presented in VII.1.

- In addition, a method defined in a class body always has a mandatory first parameter, conventionally named ***`self`***, that refers to the instance on which you call the method.

- The *`self`* parameter (also called the "calling object") is a reference to the object that call the method.

- To call a method, you first need to create an instance of the class. Then you call the method using that instance.

- Here's an example of a class that includes a method definition, and then a call to this method:

```
class Test:
    def sayHello(self, name):
        print("Hello ", name)

t=Test()                    # t is an instance of the class Test()
t.sayHello("John")          # prints      Hello John
```

I.3.4 Class-private variables

- When a statement in a class body (or in a method in the body) uses an identifier starting with two underscores (but not ending with underscores), such as *`__ident`*, the Python compiler implicitly changes the identifier into *`_classname__ident`*, where *`classname`* is the name of the class.

- This lets a class use private names for attributes, methods, global variables, and other purposes, without the risk of accidentally duplicating names used elsewhere.

- By convention, all identifiers starting with a single underscore are also intended as private to the scope that binds them, whether that scope is or isn't a class.

Note: the Python compiler does not enforce privacy conventions, however: it's up to Python programmers to respect them.

I.3.5 Class documentation strings

- Similarly to functions, if the first statement in the class body is a string literal, the compiler binds that string as the documentation string attribute for the class.

- This attribute is named **`__doc__`** and is known as the ***docstring*** of the class.

I.4 Class-Level Methods

- Class-level methods exist only in Python 2.2 and later.

I.4.1 Static methods

- A ***static method*** is a method that you can call on a class, or on any instance of the class, without the special behavior and constraints of ordinary methods, bound and unbound, on the first argument.

- A static method may have any signature and the first argument, if any, plays no special role.

- You can think of a static method as an ordinary function that you're able to call normally, despite the fact that it happens to be bound to a class attribute.

- You build a static method by calling the built-in function `staticmethod()` with as argument the method in question, and binding the value it returns to a class attribute (this is normally done in the body of the class).

```
class Test:
    def sayHello():
        print("Hello World")
    sayhello=staticmethod(sayhello)

Test.sayHello()
```

```
Hello World
```

Note: it is not mandatory to use the same name for the function passed to `staticmethod()` and for the class variable name.

- Since Python 3, the preferred (and completely equivalent) way of defining a static method is by using the ***decorator syntax***:

```
class Test:
    @staticmethod
    def sayHello():
        print("Hello World")
```


I.4.2 Class methods

- A **class method** is a method that you can call on a class or on any instance of the class.
- Python binds the method's first argument to the class on which you call the method, or the class of the instance on which you call the method (it does not bind it to the instance, as for normal bound methods).
- The first formal argument of a class method is conventionally named ***cls***.
- You build a class method by calling the built-in function **`classmethod()`** and binding its result to a class attribute.
- The only argument to `classmethod()` is the function to invoke when Python calls the class method. Here's how to define and call a class method:

```
class Test (object):
    def sayhello(cls):
        print("Hello World")
        print("Class name:", cls.__name__)
    sayhello=classmethod(sayhello)

Test.sayhello()
```

```
Hello World
Class name: Test
```

- Since Python 3, the preferred (and completely equivalent) way of defining a class method is by using the ***decorator syntax***:

```
class Test:
    @classmethod
    def sayHello():
        print("Hello World")
```

I.4.3 Function decorator

- `staticmethod()` and `classmethod()` are referred to as being **function decorator**s: a function decorator is a wrapper to an existing function.

- Python makes creating and using function decorators a bit cleaner and nicer for the programmer through some syntactic sugar: you put ***@function_name*** before the function to be wrapped.
  For instance, to decorate `f()` as a static method we can write:

```
class C:
     def f(arg1, arg2, ...): ...
     f = staticmethod(f)
```

  but there is a neat shortcut for that, which is to mention the name of the decorating function prefixed with an `@` symbol before the function to be decorated.

```
class C:
  @staticmethod
     def f(arg1, arg2, ...): ...
```

- More generally, if `dec1()` and `dec2()` are two function decorator, you can write:

```
@dec2
@dec1
def func(arg1, arg2):
    pass
```

- Which is equivalent to the "classic" syntax :

```
def func(arg1, arg2):
    pass

func = dec2(dec1(func))
```

I.5 Class Instances (Objects)

- To create an instance of a class (an object), you call the class as if it was a function. Each call returns a new instance object of that class:

```
today = Date()
```

- The built-in function **`isinstance(`*`object`*`, `*`class`*`)`** can be used to test if a given *`object`* is an instance of a given *`class`* (or any subclass of that *`class`*).

I.5.1 Constructors

- When a class has or inherits a method named **`__init__()`**, calling the class object implicitly executes `__init__()` on the new instance.

- This method should perform any instance-specific initialisation that is needed.

- Arguments passed during the creation of the instance must correspond to the formal parameters of `__init__()`.

```
class Date:
    nbOfMonths=12
    def __init__(self, d, m, y):
        self.day=d
        self.month=m
        self.year=y
    def display(self):
        print(self.day, "/", self.month, "/", self.year)

today = Date(12,3,2006)  # today is an instance of the class Date
```

- The main purpose of `__init__()` is to bind, and thus create, the **instance variables** of a newly created instance.

- When `__init__()` is absent, the newly generated instance has no instance-specific attributes (except, if any, inherited ones).

Note: the `__init__()` method does not return anything.

- A class without explicit base class, inherits `__init__()` from `object`. The `__init__()` method from `object` let you pass arbitrary arguments when you create a class instance, … but ignores all of those arguments !
  It is a good idea to override `__init__()` even in those rare cases in which your own class's `__init__()` has no task to perform.

I.5.2 Attributes of instance objects

- Once you have created an instance, you can access its attributes (instance variables and methods) using the dot (`.`)operator.

```
...
today = Date(12,3,2006)
print("day: ", today.day)
today.display()
```

- You can also give an instance object an arbitrary attribute by binding a value to an attribute reference. For example:

```
class Point: pass

center = Point( )
center.x = 10
center.y = 20
print(center.x)          # prints 10
```

- Instance object *`center`* now has two attribute named *`x`* and *`y`* bound to the value 10 and 20 and *`center.x`* and *`center.y`* refer to these attributes. But note, that this is, usually, not the recommended way of defining attributes; most of the time the attributes are defined for all instances of a given class within the special method `__init__()`.

Note: the **`__setattr__()`** special method, if present, intercepts every attempt to bind an attribute.

- Creating a class instance (defining an object, if you prefer) implicitly defines two instance attributes:

  1. **`__class__`** is the class object to which the instance belongs,
  2. **`__dict__`** is a dictionary that holds all of the other attributes of the instance.

Note: There is no difference between instance attributes created in `__init__()`, by assigning to attributes, or by explicitly binding an entry in `__dict__`.

I.5.3 Factory functions

- It is common to want to create class instances depending upon some condition.

- You can provide a function that after having tested these conditions will decide to create (or not) and return a class instance, rather than by calling the class object directly. A function used in this role is known as a ***factory function***.

- Providing a factory function is a flexible solution, as such a function may return an existing reusable instance or create a new instance by calling whatever class is appropriate.

I.5.4 Attribute Reference Basics

- An attribute reference is an expression of the form `II.name`, where `x` is any expression and `name` is an attribute name.

- Attributes that are callable are also known as ***methods***. Python draws no strong distinction between callable and non-callable attributes, as other languages do. General rules about attributes also apply to methods.

- Many kinds of Python objects have attributes, but an attribute reference has special rich semantics when `x` refers to a class or a class instance.

- When you refer to a special attribute (`__name__` , `__bases__`, `__class__`, `__dict__`), the attribute reference looks directly into a special dedicated slot in the class or instance object and fetches the value it finds there. Thus, you can never unbind these attributes.

- Rebinding them is allowed, so you can change the name or base classes of a class or the class of an instance on the fly.

- Apart from special names, when you use the syntax `II.name` to refer to an attribute of instance `x`, the lookup proceeds in two steps:

1. When `name` is a key in `II.__dict__` , the value at `II.__dict__['name']` is returned.

2. Otherwise, `II.name` delegates the lookup to `x`'s class (i.e., it works just the same as `II.__class__.name`)

- Lookup for a class attribute reference proceeds similarly.

- When these two lookup procedures do not find an attribute, Python raises an `AttributeError` exception. However, if `x`'s class defines or inherits special method `__getattr__()`, Python calls `II.__getattr__('name')` rather than raising the exception.

Note: attribute lookup steps happen only when you refer to an attribute, not when you bind or unbind an attribute.

I.5.5 The __new__() method

- Each class has a static method named `__new__()`.
- When you create a instance of a class, Python invokes its `__new__()` method. Then, Python uses `__new__()`'s return value as the newly created instance.
- Finally, Python calls the class `__init__()` method for this newly created instance.
- Thus, the statement `today=Date(23,3,2006)` is equivalent to the following code:

```
today = Date.__new__(Date, 23, 3, 2006)
Date.__init__(today, 23, 3, 2006)
```

- `object.__new__()` creates a new, uninitialized instance of the class it receives as its first argument, and ignores any other argument.
- When you override `__new__()` within the class body, Python considers it implicitly as a static method.
- If you redefine `__new__()` you can decide to make it behave as a factory function, or not. `__new__()` may choose to return an existing instance or to make a new one, as appropriate.
- When `__new__()` does need to create a new instance, it most often delegates creation by calling `object.__new__()` or the `__new__()` method of a superclass of the current class.

I.6 Instance Methods

- The class attribute reference lookup process described in the previous section actually performs an additional task when the value found is a method.

- In this case, the attribute reference does not return the method directly, but rather wraps the method into a ***bound method*** object or returns it directly as a normal ***function***.

- The key difference between functions and bound methods is that a function is not associated with a particular instance, while a bound method is.

- Using the previous defined class `Date`, here are two examples of bound method and function:

```
today = Date(12,3,2006)
print(today.display)
print(Date.display)

output:

<bound method Date.display of <__main__.Date instance at 0x009C66C0>>
<function Date.display at 0x0000000002D8FD08>
```

- We get bound methods when the attribute reference is an instance, and functions when the attribute reference is a class.

- Because a bound method is already associated with a specific instance, you call the method as follows:

```
today = Date(12,3,2006)
today.display()       # prints: 12 / 3 / 2006
```

- You don't pass the method's first argument, `self`, by the usual argument-passing syntaII. Rather, a bound method of instance `today` implicitly binds the `self` parameter to object `today`. Thus, the body of the method can access the instance's attributes as attributes of `self`, even though we don't pass an explicit argument to the method.

- A function, however, is not associated with a specific instance, so you must specify an appropriate instance as the first argument when you invoke a function.

```
today = Date(12,3,2006)
Date.display(today)          # prints: 12 / 3 / 2006
```

Note: the preferred and recommend way of using instance methods is as ***bound methods***.

- An bound method has three read-only attributes in addition to those of the function object it wraps:

    1. **`__func__`** is the wrapped function,
    2. **`__self__`** is the instance from which the method was obtained,
    3. **`__self__.__class__`** is the class object supplying the method.

- When you call a bound method, the bound method passes `__self__` as the first argument to `__func__`, before other arguments (if any) are passed at the point of call.

- Within a bound method, variables referenced are local or global, just as for any other function.

- Variables do not implicitly indicate attributes in `self`, nor do they indicate attributes in any class object. When the method needs to refer to, to bind, or to unbind an attribute of its `self` object, it does so by standard attribute-reference syntax (e.g., *`self.name`*).

I.7 Inheritance

- A class can inherit from a built-in type, an old-style class cannot.

- The object model supports multiple inheritance. However, a class may directly or indirectly subclass multiple built-in types only if those types are specifically designed to allow this level of mutual compatibility.

- Normally, a class only subclasses at most one substantial built-in type.

I.7.1 Inheritance and MRO (Method Resolution Order)

- When you use an attribute reference `ClassName.AttributeName` on a class object `ClassName`, and `AttributeName` is not a key in `ClassName.__dict__`, the lookup implicitly proceeds on each class object that is in `ClassName.__bases__`, in order.

- `ClassName`'s base classes may in turn have their own base classes. In this case, the lookup recursively proceeds up the inheritance tree, stopping when `AttributeName` is found.

- The exact order in which the bases classes will be explored depend of the version of Python you are using. To determine the exact MRO (Method Resolution Order) each class has a special read-only class attribute called **`__mro__`**, which is the tuple of types used for method resolution, in order.

Note: this resolution order is valid for both method and attributes

I.7.2 Overriding attributes

- When a subclass defines an attribute with the same name as one in a superclass, the search finds the definition when it looks at the subclass and stops there.

- This is known as the subclass **overriding** the definition in the superclass.

I.7.3 Delegating to superclass methods

- When a subclass overrides a method of its superclass, the body of the method in the subclass often wants to delegate some part of its operation to the superclass's implementation of the method.

- This can be done using an ***unbound*** method:

  ```
  BaseClassName.MethodName(self, arguments …)
  ```

- One very common use of such delegation occurs with special method `__init__()`. When an instance is created in Python, the `__init__()` methods of base classes are not automatically invoked, it is up to a subclass to perform the proper initialization by a using delegation.

```
class Date (object):
    def __init__(self, d, m, y):
        self.day=d
        self.month=m
        self.year=y
    def display(self):
        print(self.day, "/", self.month, "/", self.year)
    …
class DateTime (Date):
    def __init__(self, d, m, y, h, mn):
        Date.__init__(self, d, m, y)
        self.hour=h
        self.minute=mn
    def display(self):
        Date.display(self)
        print(self.hour, "H", self.minute)
    …

today = DateTime(12,3,2006, 13, 45)
today.display()
```

```
12 / 3 / 2006
13 H 45
```

Note: delegating to a superclass implementation is the main use of *unbound* methods.

- Python 3 provides a built-in function **`super()`** that offer facilities similar to the delegating mechanism described above.

- `super()` returns a special super-object of the calling object.

- When we look up an attribute (e.g., a method) in this super-object, the lookup start from the ancestor of the current class using the current method resolution order.

- We can therefore rewrite the previous code as:

```
class Date (object):
    def __init__(self, d, m, y):
        self.day=d
        self.month=m
        self.year=y
    def display(self):
        print(self.day, "/", self.month, "/", self.year)
    …
class DateTime (Date):
    def __init__(self, d, m, y, h, mn):
        super().__init__(d,m,y)
        self.hour=h
        self.minute=mn
    def display(self):
        super().display();
        print(self.hour, "H", self.minute)
    …

today = DateTime(12,3,2006, 13, 45)
today.display()
```

```
12 / 3 / 2006
13 H 45
```

I.7.4 "Deleting" inherited attributes

- Inheritance allows you to add or redefine class attributes (methods) without modifying the base class(es) in which the attributes are defined.

- However, inheritance does support offer similar ways to delete or hide base classes' attributes.

- If you need to perform such deletion, possibilities include:

  - Overriding the method and raising an exception in the method's body

  - Overriding `__getattribute__()` for selective delegation

I.8 The Built-in object Type

- As of Python 3, the built-in **`object`** type is the ancestor of all built-in types and custom classes.

- The `object` type defines some special methods that implement the default semantics of objects:

`__new__()`, **`__init__()`**

> You can create a direct instance of `object`, and such creation implicitly uses the static method `__new__()` of type `object` to create the new instance, and then uses the new instance's `__init__()` method to initialize the new instance.
>
> `object.__init__()` ignores its arguments and performs no operation whatsoever, so you can pass arbitrary arguments to type object when you call it to create an instance of it: all such arguments will be ignored.

`__delattr__()`, **`__getattribute__()`**, **`__setattr__()`**

> By default, an object handles attribute references as covered earlier in this chapter, using these methods of object.

`__hash__()`, **`__repr__()`**, **`__str__()`**

> See I.11.1.2.

- A subclass of object may override any of these methods and/or add others.

I. 9 Properties

- A property is an instance attribute with special functionality: when you reference, bind, or unbind a property, these accesses special methods that you specify when defining the property.

- You build a property by calling the built-in function **`property()`** and binding its result to a class attribute:

`attribute=`**`property`**`(`**`fget`**`=None, `**`fset`**`=None, `**`fdel`**`=None, `**`doc`**`=None)`

- When you reference the *`attribute`*, Python calls on the method you passed as argument **`fget`** to the `property` constructor, without arguments.

- When you assign a *`value`* to the *`attribute`*, Python calls the method you passed as argument **`fset`**, with *`value`* as the only argument.

- When you perform delete the attribute, Python calls the method you passed as argument **`fdel`**, without arguments.

- Python uses the argument you passed as `doc` as the *docstring* of the attribute.

- All arguments to property are optional. When an argument is missing, the corresponding operation is forbidden.

```
class Date (object):
    def __init__(self, d, m, y):
        self._day=d
        self._month=m
        self._year=y
    def display(self):
        print(self._day, "/", self._month, "/", self._year,
          sep="")
    def getDay(self): return self._day
    def setDay(self, n):
        if (n>0 and n <= 31):      self._day=n
        else:                      self._day=1
    def delDay(self): del self._day
    day=property(getDay, setDay, delDay, "get or set the day
of the date")

today = Date(12,3,2006)
today.day=34
today.display()
print(today.day)
```

```
1/3/2006
1
```

- You can obtain the same result using `property` as a function decorator.

```
class Date (object):
    def __init__(self, d, m, y):
        self._day=d
        self._month=m
        self._year=y
    def display(self):
        print(self._day,"/",self._month,"/",self._year,
        sep="")
    @property # <=> day = property(day)
                # day is now a special object that has extra methods:
               # getter, setter, deleter used as decorator
    def day(self):
        """
        get or set the day of the date
        """
        return self._day
    @day.setter
    def day(self, n):
        if (n>0 and n <= 31):
            self._day=n
        else:
            self._day=1
    @day.deleter
    def day(self):
        del self._day


today = Date(12,3,2006)
today.day=34
today.display()
print(today.day)
```

```
1/3/2006
1
```

I.10 __slots__

- Normally, each instance object has a dictionary `__dict__` that Python uses to let you bind arbitrary attributes on the instance.

- When a class has an attribute **`__slots__`**, an instance of the class has no `__dict__`, and any attempt to bind on the instance any attribute whose name is not in `__slots__` raises an exception.

- Using `__slots__` lets you reduce memory consumption for small instance objects that can do without the ability to have arbitrarily named attributes.

Note: `__slots__` must be defined as a class attribute.

Note: you cannot use `__slots__` with a class that uses multiple inheritance.

Note: `__slots__` does not constrain properties, only ordinary instance attributes.

I.11 Special Methods

- In Python, some names are spelled in a peculiar manner, with two leading and two trailing underscores.

- This spelling signals that the name has a special significance, and you should never invent such names for your own programs.

- One set of such names that is very prominent in the language is the set of special method names (like `__init__()`, `__new__()` and `__iter__()` for instance).

- If one of your objects implements one of these methods, that method will be called under specific circumstances (exactly which will depend on the name) by Python.

- There is rarely any need to call these methods directly.

I.11.1 General-Purpose Special Methods

I.11.1.1 Initialization and finalization

- An instance can control its initialization via the special method **`__init__(`*`self[,args]`*`)`**,

- An instance can control its finalization via the special method **`__del__(`*`self`*`)`**.

- Just before an instance disappears because of garbage collection, Python calls its **`__del__()`** method (if one is provided) to give the instance a chance to perform any cleanup action (if needed) before it cease to exist. Python performs no implicit call to `__del__()` methods of the superclasses.

I.11.1.2 Representation as string

- An instance can control how Python represents it as a string via special methods:

`__repr__(`*`self`*`)`

> This method should return a string representation of the calling object.
>
> The **`repr(`*`obj`*`)`** built-in function call `obj.__repr__()`. If `__repr__()` is absent, Python uses a default string representation.

`__str__(`*`self`*`)`

> This method should return a human-readable string, informal, concise string representation of the calling object.
>
> The **`str(obj)`** built-in type and the `print(obj)` function call `obj.__str__()` to obtain an informal, concise string representation of `obj`.
> If `__str__()` is absent, Python calls `__repr__()` instead.

I.11.1.3 Comparison

- An object instance can control how it compares with other objects

`__eq__(self,other)`
`__ge__(self,other)`
`__gt__(self,other)`
`__le__(self,other)`
`__lt__(self,other)`
`__ne__(self,other)`

The comparisons ==, >=, >, <=, <, and !=, respectively, call the special methods listed above, which should return `False` or `True`. Each method may return `NotImplemented` to tell Python to handle the comparison in alternative ways.
By default, __ne__() delegates to __eq__() and inverts the result.

Note: when neither `__eq__()` or `__ne__()` exist, == and != become identity checks: `x==y` evaluates to `id(x)==id(y)`.

I.11.1.4 Conversions into booleans

`__bool__(self)`

When evaluating an object as true or false (for example on a call to **`bool(obj)`**, Python calls `__bool__()`, which should return `True` or `False`.

When `__bool__()` is not present, Python calls **`__len__()`** instead. When neither method is present, Python always takes evaluate the object as true.

I.11.1.5 Conversion into an hash key

`__hash__(self)`

Using an object as a dictionary key calls `__hash__()`. This method must return a 32-bit `int` such that `x==y` implies `hash(x) == hash(y)`.

I.11.1.6 Attribute reference, binding, and unbinding

- An instance can control access to its attributes (reference, binding, unbinding) by defining special methods **`__getattribute__()`**, **`__getattr__()`**, **`__setattr__()`**, and **`__delattr__()`**.

- **`__delattr__(`***`self,name`***`)`** is called each time there is an attempt to unbind the attribute *name* (`del self.name`). If this method is absent, Python usually translates `del self.name` into `del self.__dict__['name ']`.

- **`__getattr__(`***`self,name`***`)`** is called each time there is an attempt to access the attribute *name* and the attribute is not present in the array `__dict__` (`__getattr__()` only gets called for attributes that don't actually exist).

- **`__getattribute__(`***`self,name`***`)`** is called each time there is an attempt to access the attribute *`name`*.

- **`__setattr__(`***`self,name,value`***`)`** is called each time there is an attempt to bind the attribute *`name`*. When `__setattr__()` binds an attribute, it must modify `__dict__` directly. If this method is absent, Python modify `__dict__` directly.

I.11.1.7 Callable instances

- An instance is callable, just like a function object, if it has the special method **`__call__(`*`self[,args]`*`)`**

- Python translates *`obj([args...])`*, into a call to *`obj.__call__([args...])`*.

I.11.2 Special Methods for Containers

- In each item access special method, a sequence that has L items should accept any integer key, such that 0<=key<L. A negative index key, 0>key>=-L, should also be accepted and equivalent to key+L.

I.11.2.1 Container special methods

`__contains__(`*`self,item`*`)`

The Boolean test **`y in x`** calls `II.__contains__(y)`.

`__delitem__(`*`self,key`*`)`

For a request to unbind an item or slice of x (typically `del` *`x[key]`*), Python will call `II.__delitem__(key)`.

`__getitem__(`*`self,key`*`)`

When *`x[key]`* is accessed Python calls *`II.__getitem__(key)`*.

`__iter__(self)`

For a request to loop on all items of x (typically *`for item in x`*), Python calls *`II.__iter__()`* to obtain an `iterator` on *`II`*.

`__len__(self)`

The *`len(x)`* built-in function calls *`II.__len__()`*.

`__setitem__(self,key,value)`

For a request to bind an item or slice of *`x`* (*`x[key]=value`*), Python calls *`II.__setitem__(key,value)`*.

I.11.2.2 Special Methods for Numeric Objects

- An instance may support numeric operations by means of many special methods.
- Some classes that are not numbers also support some of the following special methods, in order to overload operators such as + and *.
- Special methods that implement arithmetic operators:

Special Methods	Corresponding Arithmetic Operators
`__abs__(self)`	abs(self)
`__invert__(self)`	~self
`__neg__(self)`	-self
`__pos__(self)`	+self
`__add__(self,other)`	self + other
`__div__(self,other)`	self / other
`__floordiv__(self,other)`	self // other
`__mod__(self,other)`	self % other
`__mul__(self,other)`	self * other
`__sub__(self,other)`	self - other
`__truediv__(self,other)`	self / other (for non-truncating divisions)
`__divmod__(self,other)`	return a pair (quotient, remainder) equal to (self//other, self%other)
`__pow__(self,other[,modulo])`	self**other or pow(self, other)

- Special methods that implement binary operators:

Special Methods	Corresponding Binary Operators
`__and__(self,other)`	self & other
`__lshift__(self,other)`	self << other
`__or__(self,other)`	self | other
`__rshift__(self,other)`	self >> other
`__xor__(self,other)`	self ^ other

- Special methods for built-in numeric types:

Special Methods	Built-in types
`__complex__(self)`	complex(self)
`__float__(self)`	float(self)
`__int__(self)`	int(self)

- Octal and hexadecimal values:

Special Methods	Built-in functions
__hex__(self)	hex(self)
__oct__(self)	oct(self)

- Compound assignments:

Special Methods	Corresponding Assignment Operators
`__iadd__(self,other)`	self += other
`__idiv__(self,other)`	self /= other
`__ifloordiv__(self,other)`	self //= other
`__imod__(self,other)`	self %= other
`__imul__(self,other)`	self *= other
`__isub__(self,other)`	self -= other
`__itruediv__(self,other)`	self /= other
`__iand__(self,other)`	self &= other
`__ilshift__(self,other)`	self <<= other
`__ior__(self,other)`	self |= other
`__irshift__(self,other)`	self >>= other
`__ixor__(self,other)`	self ^= other
`__ipow__(self,other)`	self **= other

I.12 Metaclasses

I.12.1 What is a Metaclass

- Any Python object has a type.
- In Python, types and classes are also first-class objects. The type of a class object is also known as the class's **metaclass**.
- The object's and classes behaviors are determined largely by their type.
- The default metaclass of a newly created class is **`type`**.
- `type` is also the metaclass of all Python built-in types, including itself.
- The meta-class of a class is returned by the **`type()`** method.

I.12.2 How Python Determines a Class's Meta-class

- The metaclass of a class can be customized by passing the **`metaclass`** keyword argument in the class definition line.
- If the `metaclass` keyword is not used, and if the class has one or more base classes, it's meta-class is the metaclass of its first base class.
- If the class has no explicit ancestor, it inherits from `object` and its metaclass is `type`.

I.12.3 How a Metaclass Creates a Class

- Having determined the metaclass, Python calls the metaclass with three arguments: the class name (a string), a tuple of base classes, and a dictionary.
- The call returns the class object, which Python then binds to the class name, completing the execution of the class statement.

I.12.4 Defining and using your own metaclasses

- It's easy to define metaclasses by inheriting from `type` and overriding some methods.

- You could provide most of the features provided by metaclasses with `__new__()`, `__init__()`, `__getattribute__()`, and so on. However, a custom metaclass can be faster, since special processing is done only at class creation time, which is a rare operation.

- A custom metaclass also lets you define a whole category of classes in a framework that automatically acquires whatever interesting behavior you've coded, quite independently of what special methods the classes may choose to define.

- As an example, the predefined metaclass **`abc.ABCMeta`** is provided in order to allow the addition of Abstract Base Classes (ABCs) as "virtual base classes" to any class or type (including built-in types), including other ABCs.

I.13 Abstract Classes

- Abstract Base Classes, or ABCs are simply Python classes that are added into an object's inheritance tree to signal certain features of that object to an external inspector.

- Tests are done using `isinstance()`, and the presence of a particular ABC means that the test has passed.

- In addition, the ABCs define a minimal set of methods that establish the characteristic of the type. Code that discriminates objects based on their ABC type can trust that those methods will always be present.

- The module **`abc`** serves as an "ABC support framework" appears in Python 3. It defines a metaclass for use with ABCs (**`ABCMeta`**) and a decorator that can be used to define abstract methods and properties (**`@abstractmethod`** and **`@abstractproperty`**).

- A class containing at least one method or property declared with one of this decorator that hasn't been overridden yet cannot be instantiated.

```
from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    def bar(self): print("Test")
    @abstractmethod
    def foo(self): pass

A()  # raises TypeError

class B(A):
    pass

B()  # raises TypeError
```

- There are two ways to indicate that a concrete class implements an abstract class: either create a new subclass directly from the abstract base or explicitly register the class using the **`register()`** decorator.

```
class C(A):
    def foo(self): print(42)

C()  # works
C().foo()  # works
C().bar()  # works
```

```
@A.register
class D:
    def foo(self): print(42)

D()  # works
D().foo()  # works
D().bar()  # Exception raised: no attribute bar()
```

- Unlike Java's or C++'s abstract methods, Python abstract methods may have an implementation. Such methods may be called from the overriding method in the subclass (using `super()` or direct invocation).

Numeric abstract classes

- Python 3 defines a hierarchy of Abstract Base Classes to represent number-like classes:

  `Number` <- **`Complex`** <- **`Real`** <- **`Rational`** <- **`Integral`**

II MODULES

- Python **modules** can be used to encapsulate related functions and other Python objects together, save them to a file, and import these already-defined functions and objects into any new Python code, including into a Python interpreter.

II.1 Module Objects

- A module is a Python object with arbitrarily named attributes that you can bind and reference.

- The Python code for a module named *aname* normally resides in a file named *aname.py*.

- To use a module in a Python program, you use the **`import`** statement:

  `import` ***`modname`*** **`[as`** ***`varname`*****`][,...]`**

- The `import` keyword is followed by one or more module specifiers, separated by commas.

- In the simplest and most common case, *modname* is an identifier, the name of a variable that Python binds to the module object when the import statement finishes.

- *modname* can also be a sequence of identifiers separated by dots (`.`) that names a module in a package (see II.11).

- When as *varname* is part of an `import` statement, Python binds the variable named *varname* to the module object, but the module name that Python looks for is *modname*.

Note: modules being objects you can pass a module as an argument in a call to a function. and similarly, a function can return a module as the result of a call. A module, just like any other object, can be bound to a variable, an item in a container, or an attribute of an object.

II.2 Module body

- The body of a module is the sequence of statements in the module's source file. There is no special syntax required to indicate that a source file is a module; any valid source file can be used as a module.

- A module's body executes immediately the first time the module is imported in a given run of a program.

- During execution of the body, the module object already exists and an entry in **`sys.modules`** is already bound to the module object.

II.3 Attributes of module objects

- An `import` statement creates a new namespace that contains all the attributes of the module.

- To access an attribute in this namespace, you use the name of the module object as a prefI.

- For instance, to call a function defined within a module, you must prefix its name with the name of the module it belongs. The complex name is necessary due to scope, which refers to the visibility of names in a program. Putting things in a module makes them have the scope of that module.

```
>>> import test
>>> test.stats([1, 2, 3, 4, 5])
3.0
```

- The **`from`** statement can be used to implicitly bring the listed method into the current file's scope, allowing you to call the function directly (without using the module's name).

  `from modname import attrname [as varname][,...]`
  `from modname import *`

- A `from` statement specifies a module name, followed by one or more attribute specifiers separated by commas.

```
>>> from test import stats
>>> stats([1, 2, 3, 4, 5])
3.0
```

- Code that is directly inside a module body (not in the body of a function or class) may use an asterisk (**`*`**) in a `from` statement:

  `from modname  import *`

- The **`*`** requests that all attributes of module *`modname`* be bound as global variables in the importing module.

- When the module *`modname`* has an attribute named **`__all__`**, the attribute's value is the list of the attributes that are bound by this type of from statement. Otherwise, this type of from statement binds all attributes of *`modname`* except those beginning with underscores.

- When a statement in the body binds a variable (a **global variable**), what gets bound is an attribute of the module object.

- In Python, global variables are not global to all modules, but instead such variables are attributes of a single module object.

- You can also bind and unbind module attributes outside the body (i.e., in other modules), generally using attribute reference syntax `M.name` (where `M` is any expression whose value is the module, and identifier `name` is the attribute name).

II.3.1 Module-private variables

- No variable of a module is really private. However, by convention, starting an identifier with a single underscore (_), indicates that the identifier is meant to be private.

II.4 Implicit module attributes

- The `import` statement implicitly defines some module attributes as soon as it creates the module object, before the module's body executes.

- The **`__dict__`** attribute is the dictionary object that the module uses as the namespace for its attributes. All other attributes in the module are entries in the module's `__dict__`, and they are available to code in the modules as global variables.

- Attribute **`__name__`** is the module's name, and attribute **`__file__`** is the filename from which the module was loaded, if any.

II.5 Module libraries

- A principal benefit of using the Python programming language is the large built-in standard library, accessible as Python modules.

- Examples of commonly used modules include:
  - **`math`** contains useful mathematical functions.
  - **`sys`** contains data and methods for interacting with the Python interpreter.
  - **`array`** contains array datatypes and related functions.
  - **`datetime`** contains useful date and time manipulation functions.
  - …

- Because these are built-in modules, you can use the help interpreter to learn more about them.

- You can import multiple modules into a Python program.

```
>>> from math import sqrt
>>> from test import stats
>>> v=stats([1, 2, 3, 4, 5])
>>> print(v, sqrt(v))
3.0 1.73205080757
```

- The loading operation takes place only the first time a module is imported in a given run of the program. When a module is imported again, the module is not reloaded.

- Currently loaded modules are listed in the dictionary **`sys.modules`**.

II.6 The __builtin__ module

- Python offers several built-in objects and functions.
- All built-in objects and functions are attributes of a preloaded module named **`__builtin__`**.
- When Python loads a module, the module automatically gets an extra attribute named `__builtins__`, which refers to the module `__builtin__`.
- When a global variable is not found in the current module, Python looks for the identifier in the current module's `__builtins__` before raising `NameError`.

II.7 Module documentation strings

- If the first statement in the module body is a string literal, the compiler binds that string as the module's documentation string attribute, named **`__doc__`**.

II.8 Searching the Filesystem for a Module

- If module *`M`* is not built-in `import` looks for *`M`*'s code as a file on the filesystem.
- `import` looks in the directories whose names are the items of list **`sys.path`**, in order.
- `sys.path` is initialized at program startup, using environment variable **`PYTHONPATH`** if present.
- Your code can mutate or rebind `sys.path`.
- When looking for module Python considers the following extensions in the order listed:

    1. `.pyd` and `.dll` (Windows) or `.so` (most Unix-like platforms), which indicate Python extension modules.
    2. `.py`, which indicates pure Python source modules.
    3. `.pyc` (or `.pyo`, if Python is run with option `-O`), which indicates bytecode-compiled Python modules.

II.9 The Main Program

- Execution of a Python application normally starts with a top-level script (also known as the main program).

- The main program executes like any other module being loaded except that Python keeps the bytecode in memory without saving it to disk.

- The module name for the main program is always **`__main__`**.

- Code in a Python module can test whether the module is being used as the main program by checking if global variable `__name__` equals `'__main__'`.

- The idiom:

```
if __name__=='__main__':
   ...
```

  is often used to guard some code so that it executes only when the module is run as the main program.

II.10 The reload Function

- As explained earlier, Python loads a module only the first time you import the module during a program run.

- When you develop interactively, you need to make sure that your modules are reloaded each time you edit them (some development environments provide automatic reloading).

- To reload a module, pass the module object (not the module name) as the only argument to the function **`reload()`** (`reload()` belongs to the package **`importlib`**).

II.11 Packages

- A **package** is a module that contains other modules.

- Modules in a package may be subpackages, resulting in a hierarchical tree-like structure. A package named `P` resides in a subdirectory, also called `P`, of some directory in `sys.path`.

- The module body of `P` is in the file `P/__init__.py`. You must have a file named `P/__init__.py`, even if it's empty (representing an empty module body), in order to indicate to Python that directory `P` is indeed a package.

- Other .py files in directory `P` are the modules of package `P`. Subdirectories of `P` containing `__init__.py` files are subpackages of P. Nesting can continue to any depth.

- When a package is imported, this `__init__.py` file is implicitly executed, and the objects it defines are bound to names in the package's namespace.

- You can import a module named `M` in package `P` as:

```
import P.M
```

- More dots let you navigate a hierarchical package structure.

- An alternative way of importing the submodule is:

```
from P import M
```

  this also loads the submodule `M`, and makes it available without its package prefix,

- If a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered.

- If `__all__` is not defined, the statement `from P import *` does not import all submodules from the package `P` into the current namespace; it only ensures that the package `P` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package.

- The simplest, cleanest way to share objects (such as functions or constants) among modules in a package P is to group the shared objects in a file named `P/Common.py`. Then you can import `Common` from every module in the package that needs to access the objects, and then refer to the objects as `Common.f`, `Common.K`, and so on.

III INDEX

P

R

S

T