

I INTRODUCTION

I.1 Overview

- Python is a general-purpose programming language.
- **Guido van Rossum**, Python's creator, started developing Python back in 1990.
- This stable and mature language is very high level, interpreted, dynamic, object-oriented, and cross-platform.
- Python runs on all major hardware platforms and operating systems, so it doesn't constrain your platform choices.
- Python offers high productivity for all phases of the software life cycle: analysis, design, prototyping, coding, testing, debugging, tuning, documentation, deployment, and, of course, maintenance.
- You'll quickly become productive with Python, thanks to its consistency and regularity, its rich standard library, and the many other modules that are readily available for it.
- Python is easy to learn, so it is quite suitable if you are new to programming, yet at the same time it is powerful enough for the most sophisticated expert.
- Even though Python might not be as fast as compiled languages such as C or C++, what you save in programming time will probably make Python worth using; in most programs the speed difference won't be noticeable anyway.

I.2 The Python Standard Library and Extension Modules

- The standard Python library and other extension modules are almost as important for effective Python use as the language itself.
- The **Python standard library** supplies many 100% pure Python modules.
- It includes modules for such tasks as data representation, string and text processing, interacting with the operating system and filesystem, and web programming. Because these modules are written in Python, they work on all platforms supported by Python.
- **Extension modules**, from the standard library, let Python applications access functionality supplied by the underlying operating system or other software components, such as graphical user interfaces (GUIs), databases, and networks.
- Extension modules that are not coded in Python, however, do not necessarily enjoy the same cross-platform portability as pure Python code.
- You can write special-purpose extension modules in lower-level languages to achieve maximum performance. You can also use tools such as `SWIG` to make existing C/C++ libraries into Python extension modules.

I.3 Python development

- Python is developed by the **Python Labs of Zope Corporation**, which consists of half a dozen core developers headed by Guido van Rossum.
- Python intellectual property is vested in the **Python Software Foundation** (PSF), a non-profit corporation devoted to promoting Python.
- This course material focuses on **Python 3.x** (in 2017, the available versions are 2.7 and 3.6).

I.4 How to run a Python program

- In many, if not most, Linux and UNIX installations, a Python interpreter will already be present.
- You can check whether this is the case by running the **`python`** command at the prompt, as follows:

```
$ python
```

- Running this command should start the interactive Python interpreter, with output similar to the following:

```
Python 2.7.3 (#1, May 7 2012, 09:18:58)
[GCC 3.4.1] on sunos5
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

- The string >>> is a Python prompt, which indicates that Python is ready to execute your commands. To exit the interactive interpreter, use `Ctrl-D`.
- The Python interpreter command-line syntax can be summarized as follows:

`python [options] [ -c command | file | - ] [arguments]`

- Possible options are **`-V`** (version), **`-h`** (help), **`-t`** (warn about inconsistent use of tab), …
- You can use **`-c`** *`command`* to execute a Python code string command.
- You can also provide the *`file`* path of a Python source or bytecode file to run.
- A hyphen, or the lack of any token in this position, tells the interpreter to read program source from standard input (and enter in an interactive mode).
- The Python distribution normally comes with a simple IDE (Integrated Development Environment) named **`idle`**.

I.5 Environment Variables

- The **`python`** program use several environment variables:

`PYTHONHOME`
: The Python installation directory. A `lib` subdirectory, containing the standard Python library modules, should exist under this directory.

`PYTHONPATH`
: This is a list of directories, separated by colons, from where Python modules are imported. This extends the initial value for Python's `sys.path` variable.

`PYTHONSTARTUP`
: The name of a Python source file that is automatically executed each time an interactive interpreter session starts.

I.6 Running Python Programs

- A Python application, can be seen as a set of Python source files.

- A **script** is a file that you can run directly.

- A **module** is a file that you can import to provide functionality to other files or to interactive sessions.

Note: a Python file can be both a module and a script, exposing functionality when imported, but also suitable for being run directly (see X.9).

- The Python interpreter automatically compiles Python source files as needed.

- Python source files normally have extension **`.py`**. Python saves the compiled bytecode file for each module in the same directory as the module's source, with the same basename and extension **`.pyc`** (or **`.pyo`** if Python is run with option **`-O`**).

Note: Python saves bytecode files only for modules you import.

- From Unix/Linux you can invoke python directly, from a shell script, or in a command file.

- On these systems, you can make a Python script directly executable by setting the file's permission bits **`x`** and **`r`** and beginning the script with a first line of the form:

```
#!/usr/bin/env python-path {options}
```

```
#!/usr/bin/env python

def stats(data):
    total = 0
    for value in data:
        total += value

    mean = total/len(data)

    total = 0
    for value in data:
        total += (mean - value)**2

    variance = total/len(data)

    return(mean, variance)

values=[1, 2, 3, 4, 5, 6, 7, 8, 9]

m, v = stats(values)
print("The mean and variance of the values",
"from 1 to 9 inclusive are ",m, "and", v)
```

- To run the statements in this file, you must enter the example above into a file named like *pyprog.py*, make the file executable (using the `chmod` UNIX/Linux command) and execute the file.

```
$ chmod +x pyprog.py
$ ./pyprog.py
…
```

Note: if the `PATH` variable includes the directory `.` , you can execute the Python program like this:

```
$ pyprog.py
```

- Another possibility is to call explicitly the python interpreter and pass to it, as an argument, the name of the python file.

```
$ python pyprog.py
…
```

II OVERVIEW OF THE PYTHON SYNTAX

II.1 Lexical Structure of a Python program

II.1.1 Lines and Indentation

- A Python program is composed of a sequence of logical lines, each made up of one or more physical lines.
- Each physical line may end with a ***comment***.
- A pound sign (**#**) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment.
- A line containing only whitespaces, possibly with a comment, is called a blank line, and is ignored by the interpreter.
- In an interactive interpreter session, you must enter an empty physical line (without any whitespace or comment) to terminate a multiline statement.
- In Python, the end of a physical line marks the end of most statements.
- When a statement is too long to fit on a single physical line, you can join two adjacent physical lines into a logical line by ensuring that the first physical line has no comment and ends with a backslash (****).
- Python also joins adjacent physical lines into one logical line if an open parenthesis ((), bracket ([), or brace ({) has not yet been closed.
- Triple-quoted string literals (see IV.3.1) can also span physical lines.
- Python uses **indentation** to express the **block** structure of a program (unlike C or C++, Python does not use braces around blocks of statements).
- Each logical line in a Python program is indented by the whitespace on its left.
- A block is a contiguous sequence of logical lines, all indented by the same amount; the block is ended by a logical line with less indentation.
- The first statement in a source file must have no indentation.
- It is recommended not to mix spaces and tabs for indentation, since different tools treat tabs differently. The **`-t`** option of the Python interpreter ensure against inconsistent tab and space usage in Python source code.

II.1.2 Tokens

- Python breaks each logical line into a sequence of elementary lexical components, called tokens: identifiers, keywords, operators, delimiters, literals ….

- Whitespace may be freely used between tokens to separate them.

II.1.3 Identifiers

- An identifier is a name used to identify a variable, function, class, module, or other object.

- An identifier starts with a letter (A to Z or a to z) or underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

- Python is case sensitive.

- Python includes 33 keywords which are identifiers that Python reserves for special syntactic uses. It is forbidden to use one of these keywords as an identifier.

`and`	`del`	`global`	`not`	`yield`
`as`	`elif`	`if`	`or`	`False`
`assert`	`else`	`import`	`pass`	`None`
`break`	`except`	`in`	`raise`	`True`
`class`	`finally`	`is`	`return`	
`continue`	`for`	`lambda`	`try`	
`def`	`from`	`nonlocal`	`while`	

II.1.3.1 Naming conventions

- Normal Python style is to start class names with an uppercase letter and other identifiers with a lowercase letter.

- Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private.

- Starting an identifier with two leading underscores indicates a strongly private identifier; if the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

- The identifier **_** (a single underscore) is special in interactive interpreter sessions: the interpreter binds _ to the result of the last expression statement evaluated interactively, if any.

II.1.4 Simple statements

- A simple statement is one that contains no other statements.
- A simple statement lies entirely within a logical line.
- You may place more than one simple statement on a single logical line, with a semicolon (`;`) as the separator.
- Any expression can stand on its own as a simple statement.
- Simple statements perform a single operation: a variable initialization, a method invocation, and a multiplication operation, …
- The interactive interpreter shows the result of an expression statement entered at the prompt (>>>), and also binds the result to a variable named `_` .
- An assignment is a simple statement that assigns a value to a variable. Unlike in C or C++, an assignment in Python is a statement, and therefore can never be part of an expression.

```
>>> i1 = 10 ; i2 = 20 ; i3 = 30
>>>
>>> b = ((i1 < 20) and
...       (i2 < 30) and
...       (i3 < 40))
>>> b
True
>>>
>>> b = (i1 < 20) and \
...      (i2 < 30) and \
...      (i3 < 40)
>>>
>>> b
True
```

II.1.5 Compound statements

- A compound statement contains other statements and controls their execution.
- A compound statement has one or more ***clauses***, aligned at the same indentation.
- Each clause has a header that starts with a keyword and ends with a colon (`:`), followed by a body, which is a sequence of one or more statements.
- When the body contains multiple statements, also known as a ***block***, these statements should be placed on separate logical lines after the header line and indented rightward from the header line.

```
simple statement one
compound statement one:
    simple statement two
    simple statement three
    compound statement two:
        simple statement four
simple statement five
```

- Alternatively, the body can be a single simple statement, following the `:` on the same logical line as the header.
- The body of a Python compound statement cannot be empty: it must contain at least one statement. The **`pass`** statement, which performs no action, can be used as a placeholder when a statement is syntactically required but you have nothing specific to do.

II.1.6 Encoding of a Python source file

- The **Unicode** standard describes how characters are represented by **code points**. A code point is an integer value, usually denoted in base 16.
- A Unicode string is a sequence of code points, this sequence needs to be represented as a set of bytes (meaning, values from 0 through 255) in memory. The rules for translating a Unicode string into a sequence of bytes are called an encoding.

- **UTF-8** is one of the most commonly used encodings. UTF stands for "Unicode Transformation Format", and the '8' means that 8-bit numbers are used in the encoding. (There are also a UTF-16 and UTF-32 encodings, but they are less frequently used than UTF-8.) UTF-8 uses the following rules:
  1. If the code point is < 128, it's represented by the corresponding byte value.
  2. If the code point is >= 128, it's turned into a sequence of two, three, or four bytes, where each byte of the sequence is between 128 and 255.

- Since Python 3.0, the language features a **`str`** type that contain Unicode characters and the default encoding for Python source code is UTF-8 (you can simply include a Unicode character in a string literal or use Unicode characters in identifiers).

- You can use a different encoding from UTF-8 by putting a specially-formatted comment as the first or second line of the source code:

  `# -*- coding: `*`encoding-name`*` -*-`

  More precisely, the first or second line of the script must match the regular expression

  ```
  coding[:=]\s*([-\w.]+)
  ```

  The first group of this expression is then interpreted as encoding name (utf-16, iso-8859-1, ...).

```
# -*- coding: utf-16 -*-
```

Note: the default encoding for Python 2.x source code is ASCII.

II.2 Variables and other References

- A Python program accesses data values through references.
- A reference is a name that refers to the specific location in memory of a value.
- References take the form of ***variables***, ***attributes***, and ***items***.
- In Python, a variable or reference has no intrinsic type. The object to which a reference is bound at a given time does have a type, however.
- Any given reference may be bound to objects of different types during the execution of a program.

II.2.1 Variables

- A variable is basically a name that represents (or refers to) some value.
- In Python, there are no declarations.
- To create a variable, you can simply execute the following:

```
>>> x = 3
```

- This is called an ***assignment***.
- We assign the value 3 to the variable x. Another way of putting this is to say that we ***bind*** the variable x to the value (or object) 3.
- After a variable has had a value assigned to it, you can use the variable in expressions:

```
>>> x * 2
6
```

Note: you have to assign a value to a variable before you use it.

II.2.2 Binding, rebinding

- The existence of a variable depends on a statement that ***binds*** the variable, or, in other words, that sets a name to hold a reference to some object.
- Assignment statements are the most common way to bind variables and other references.
- Binding a reference that was already bound is also known as ***rebinding*** it.

II.2.3 Unbinding

- You can ***unbind*** a variable by resetting its name so it no longer holds a reference.
- The **`del`** statement can also be used to unbind references.
- A `del` statement consists of the keyword **`del`**, followed by one or more target references separated by commas (,).
- Each target can be a variable, attribute reference, indexing, or slicing, just like for assignment statements, and must be bound at the time `del` executes.
- Despite its name, a `del` statement does not delete objects: rather, it ***unbinds*** references. Object deletion may follow as a consequence, by garbage collection, when no more references to an object exist.
- When the `del` statement specifies a request to an object to unbind one or more of its attributes or items. An object may refuse to unbind some (or all) attributes or items, raising an exception if a disallowed unbinding is attempted (see `__delattr__()`).

II.2.4 Garbage collection

- Rebinding or unbinding a reference has no effect on the object to which the reference was bound, except that an object ceases to exist when nothing refers to it.
- The automatic clean-up of objects to which there are no references is known as **garbage collection**.

II.2.5 Global and local variables

- A variable can be **global** or **local**.
- A global variable is an attribute of a module object (see X.3)
- A local variable lives in a function's local namespace.

II.2.6 Object attributes and items

- An **attribute** of an object is denoted by a reference to the object, followed by a period (**.**), followed by an identifier called the attribute name (i.e., x.y refers to the attribute of object x that is named y).
- An **item** of an object is denoted by a reference to the object, followed by an expression within brackets ([ ]). The expression in brackets is called the *index* or *key* to the item, and the object is called the container of the item (i.e., x[y] refers to the item at key or index y in container object x).

II.2.7 Assignment Statements

- Assignment statements can be simple or compound.
- Simple assignment use the **=** operator.

  target = expression

- When the assignment statement executes, Python evaluates the right-hand side expression, then binds the expression's value to the left-hand side target.

II.2.7.1 Simple assignment

- Simple assignment to a variable (e.g., *name=value*) is how you create a new variable or rebind an existing variable to a new value.
- When the target of the assignment is an identifier, the assignment statement specifies the binding of a variable. This is never disallowed: when you request it, it takes place. In all other cases, the assignment statement specifies a request to an object to bind one or more of its attributes or items. An object may refuse to create or rebind some (or all) attributes or items, raising an exception if you attempt a disallowed creation or rebinding.
- There can be multiple targets and equals signs (=) in a plain assignment. For example:

  ```
  a = b = c = 0
  ```

  binds variables a, b, and c to the value 0. Each time the statement executes, the right-hand side expression is evaluated once. Each target gets bound to the single object returned by the expression, just as if several simple assignments executed one after the other.
- The target in a plain assignment can list two or more references separated by commas, optionally enclosed in parentheses or brackets. For example:

  ```
  a, b, c = x
  ```

  This requires x to be a sequence with three items, and binds `a` to the first item, `b` to the second, and `c` to the third.

Note: with Python 3.0 came the ***extended iterable unpacking*** which brought an elegant way to extract needed data into variables and also specify a "catch-all" variable that will take the rest:

```
data = [1, 2, 3, 4]
a, b, *rest = data
# a=1, b=2, rest=[3, 4]
```

II.2.7.2 Compound assignment

- A compound assignment differs from a plain assignment in that, instead of an equals sign (=), it uses a compound operator: a binary operator followed by =.

- The compound operators are **+=**, **-=**, ***=**, **/=**, **//=**, **%=**, ****=**, **|=**, **>>=**, **<<=**, **&=**, and **^=**.

- A compound assignment never creates its target reference: the target must already be bound when compound assignment executes.

- A compound assignment can rebind a variable, or rebind an object attributes or items.

- A compound assignment can have only one target on the left-hand side.

- In a compound assignment, Python first evaluates the right-hand side expression. Then, if the left-hand side refers to an object that has a special method for the appropriate in-place version of the operator, Python calls the method with the right-hand side value as its argument. It is up to the method to modify the left-hand side object appropriately and return the modified object.

- If the left-hand side object has no appropriate in-place special method, Python applies the corresponding binary operator to the left-hand side and right-hand side objects, and then rebinds the target reference to the operator's result.
  For example, `x += y` is like `x = x.__iadd__(y)` when `x` has special method `__iadd__()`. Otherwise `x+=y` is like `x=x+y`.

II.3 Standard Input

- The **`sys`** module provides the **`stdin`** attribute, which is a `file` object from which you can read text entered via the keyboard (by default).

- When you need a line of text from the user, you can call the built-in function **`input()`**, optionally with a string argument to use as a prompt.

  `input(prompt='')`

- This function writes the *`prompt`* to standard output, reads a line from standard input, and returns the line (without `\n`) as a `string`.

Note: in Python 2.x `raw_input()` is similar to `input()`.

Note: in Python 2.x, when the input you need is not a string (for example, when you need a number), you can use the built-in function `input()`. `input(prompt)` is a shortcut for `eval(raw_input(prompt))`. In other words, input prompts the user for a line of input, evaluates the resulting string as an expression, and returns the expression's result. The implicit `eval()` makes the `input()` function very vulnerable.

II.4 The print function

- In interactive use of the Python interpreter, you can type an expression and immediately see the result of its evaluation. In a program run from a file Python does not display expressions this way. If you want your program to display something, you can give explicit instructions with the **`print()`** function.

```
x = 3
y = 5
print("The sum of", x, "plus", y, 'is', x+y)
```

- The `print()` function will prints as strings every element of a comma-separated sequence of expressions, and it will separate each element with a single whitespace by default.
  Any expression that is not already a string is automatically converted to its string representation using the built-in **`str()`** function.

- Python has 2 ways to convert any value to a string: pass it to the **`repr()`** or `str()` functions.

- The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

- You can control output format more precisely by performing string formatting (see IV.3.4).

- `print()` can also be used with no parameters to just print a new line.

- The `print()` function also accepts a certain amount of *keyword arguments*:

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

- It is possible to redefine the separator between printed expressions by assigning an arbitrary string to the keyword parameter "**`sep`**" :

```
>>> print("a","b")
a b
>>> print("a","b",sep="")
ab
```

- A print call is ended by a newline, to change this behaviour we can assign an arbitrary string to the keyword parameter "**`end`**". This string will be used for ending the output of the values of a print call:

```
>>> for i in range(4):
...     print(i, end=" ")
...
0 1 2 3
```

- The output of the print function is send to the standard output stream (`sys.stdout`) by default. By using the keyword parameter "**`file`**" we can send the output to a different stream e.g. `sys.stderr` or a file.

- Whether output is buffered is usually determined by the file used by `print()`, but if the **`flush`** keyword argument is true (it is False by default), the stream is forcibly flushed.

```
>>> fh = open("data.txt","w")
>>> print("42 is the answer, but what is the question?",
file=fh, flush=True)
>>> fh.close()
```

Note: Python 2.x do have a print statement denoted by the keyword **`print`** (the `print()` function is not available by default)

```
nb=12
print "Nb is", nb, "..."           # prints: Nb is 12 ...
```

III THE PYTHON TYPE HIERARCHY
III.1 Everything is an object

- One of the most important point to learn when transitioning to the Python programming language from another language is that **everything in Python is an object**.

- This might not seem unusual, especially if you're familiar with an object-oriented language, such as C++, Java, or C#. But Python's object-oriented philosophy goes beyond that of these other languages, as evidenced by two simple differences. First, all data values in Python are encapsulated in relevant object classes. Second, everything in a Python program is an object accessible from within your program, even the code you write.

- Most popular programming languages have several built-in data types, and Python is no different in this respect. But unlike Java or C++, Python, built-in data types are object types.

- If you need an integer value in Python, you merely assign an integer value to the appropriate variable, such as `i = 100`. Under the covers, Python creates an integer object and assigns the variable to reference the new object.

- If needed, Python makes it easy to determine a variable's type, using the **`type()`** function.

```
>>> i = 100 # Creates an int object whose value is 100
>>> type(i)
<type 'int'>
>>> f = 100.0
>>> type(f)
<type 'float'>
```

- Python is a ***dynamically typed language***, so you don't have to declare a variable's type. In fact, a variable's type can actually change (multiple times) during a single program.

- An easy way to visualize how the dynamic typing works is to imagine a single base class called **`object`** from which all other object types in Python inherit. In this model, any variable you create references an object that has been created from the overall class hierarchy. If you also have the `object` class record the actual type, or name, of the child class, which is created and assigned to the variable, a Python program can properly determine the necessary steps to take during program execution.

- The mental picture presented in the previous paragraph describing Python's object-oriented model is a fairly good approximation of how Python actually works.

- You can classify all the Python classes below the `object` class into four main categories that the Python run-time interpreter uses:

    1. **Simple types** -- The basic building blocks, like int and float
    2. **Container types** -- Hold other objects
    3. **Code types** -- Encapsulate the elements of your Python program
    4. **Internal types** -- Used during program execution

III.2 The simple types

- Python has five simple built-in types: **`bool`**, **`int`**, **`float`**, and **`complex`**.

Note: Python 2.x do also offer the **`long`** built-in type. `long` in Python 2 is similar to `int` in Python 3.

- These types are ***immutable***, which means, for instance, that when an integer object is created, its value cannot be changed. Instead, a new simple type object is created and assigned to the variable. Using the Python **`id()`** function, you can see how the identification of the underlying object changes:

```
>>> i = 100
>>> id(i)
8403284
>>> i = 101
>>> id(i)
8403296
```

Note: as stated previously, to avoid memory leak, python, like C# and the Java language, employs a ***garbage collector*** that frees up memory used to hold objects that are no longer referenced, like the integer object that holds 100 in the previous example.

III.3 The Boolean type

- The simplest built-in type in Python is the **`bool`** type, which can hold only one of two possible predefined objects: **`True`** or **`False`**:

```
>>> b = True
>>> type(b)
<type 'bool'>
>>> id(b)
1041552
```

- The case for the name of the Boolean object is important (`true` and `false` are undefined).

- Because there are only two possible values, the Boolean type is unique. The Python interpreter provides the only two `bool` objects needed: `True` and `False`. Anytime these objects are needed in a Python program, the variable just references one as appropriate. The following example shows how the *`bb`* variable has the same id whether you assign it a value of the *`b`* variable directly or just the `True` object directly.

```
>>> b = True
>>> id(b)
1041552
>>> bb = b
>>> id(bb)
1041552
>>> bb = True
>>> id(bb)
1041552
```

- Many programs utilize Boolean expressions, and Python provides the full range of Boolean comparison and logical operations, respectively.

Comparison operators	**Description**	**Example**
<	less than	i < 100
<=	less than or equal to	i <= 100
>	greater than	i > 100
>=	greater than or equal to	i >= 100
==	equality	i == 100
!=	inequality	i != 100

- To be complete, all the operators listed in the above table have equal precedence and, unless you enclose expressions in parentheses, are applied from left to right.

Logical Operator	**Description**	**Example**
not	logical negation	not b
and	logical and	(i <= 100) and (b == True)
or	logical or	(i < 100) or (f > 100.1)

- The logical operators have lower precedence than the individual comparison operators (the comparison has to be evaluated before the logical operator can be evaluated). The actual precedence of the logical operators is given by the order these operators are presented in above table.

- One interesting point about the **or** and the **and** logical operators in Python is that they are both shortcut operators. In simple terms, this means that given `x or y`, *y* is evaluated only if *x* is `False`. Likewise, given the expression `x and y`, *y* is evaluated only if *x* is `True`.

- In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, 0, and empty strings and containers. All other values are interpreted as true.

III.4 The numeric types

- The other four simple, built-in types in Python are all numeric types: **`int`**, **`float`**, and **`complex`**.

- The Arithmetic operators provided by Python are listed into the following table, in their precedence order:

Operator	Description
`x ** y`	x to y^{th} power
`x * y`	Multiplication
`x / y`	Division
`x // y`	Truncating division: returns an integer result (converted to the same type as the wider operand) and ignores the remainder, if any.
`x % y`	The modulo operator yields the remainder from the division of x by y.
`+y`	Unary +
`-y`	Unary -
`x + y`	Addition
`x - y`	Subtraction

- You can group sub-expressions and give them higher precedence by using parentheses to set them apart.

- **`float`** map to IEEE-754 "double precision" and floating-point numbers are represented internally as base 2 (binary) fractions. Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. This is the chief reason why Python (but also C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect.

- For use cases which require exact decimal representation, you can try using the decimal module which implements decimal arithmetic suitable for accounting applications and high-precision applications.

- Another form of exact arithmetic is supported by the fractions module which implements arithmetic based on rational numbers.

- Finally, if you are a heavy user of floating point operations you should take a look at the Numerical Python package and many other packages for mathematical and statistical operations supplied by the SciPy project.

- The **`int`** integer type has unlimited precision (only subject to the memory limitations of your computer).

Note: with Python 2.x you can use both the **`int`** and **`long`** types to hold integer values, the difference being that an `int` is a 32-bit integer value. Thus, it is limited to holding values (on many platforms) from -2^{32} to $2^{32} - 1$.
In contrast, the **`long`** integer type has unlimited precision.
To tell Python that an integer should be treated as a long, simply append an `L` to the end of the number, like *`100L`*.

- Python provides support for decimal (the default), binary (base 2), octal (base 8) and hexadecimal (base 16) numeric literals. Different 2 characters prefixes have to be used to tell Python that a number should be treated as binary ('**0b**'), octal ('**0o**') or hexadecimal ('**0x**') numeric literals:

```
>>> print(127)     # Using decimal literal
127
>>> print(0o177)   # Using octal literal
127
>>> print(0x7F)    # Using hexadecimal literal
127
>>> print(0b1111111)    # Using a binary literal
127
```

III.4.1 Bitwise Operations on Integers

- Integers and long integers can be considered strings of bits and used with the bitwise operations shown in the following table.

- Bitwise operators have lower priority than arithmetic operators.

- Positive integers are extended by an infinite string of 0 bits on the left.

- Negative integers are represented in two's complement notation, and therefore are extended by an infinite string of 1 bits on the left.

Operator	Description
~x	bitwise not
x **>>** y x **<<** y	Logical shift right Logical shift left
x **&** y	Bitwise AND
x **^** y	Bitwise XOR
x **|** y	Bitwise OR

- The << operator is used to perform a signed left shift. The value of `n<<s` is `n` left-shifted `s` bit positions.

```
>>> i = 6
>>> i = i << 3
>>> i
48
```

0	0	0	0	0	1	1	0
0	0	1	1	0	0	0	0

- The >> operator is used to perform a signed right shift. The value of `n>>s` is `n` right-shifted `s` bit positions with sign-extension.

```
>>> i = -14
>>> i = i >> 1
>>> i
-7
```

1	1	1	1	0	0	1	0
1	1	1	1	1	0	0	1

- For **&**, the result value is the bitwise AND of the operand values.

&	0	1
0	0	0
1	0	1

```
>>> i= 14
```

0	0	0	0	1	1	1	0

```
>>> j= 7
```

0	0	0	0	0	1	1	1

```
>>> res= i & j
>>> res
6
```

0	0	0	0	0	1	1	0

- For **^**, the result value is the bitwise exclusive OR of the operand values.

^	0	1
0	0	1
1	1	0

- For **|**, the result value is the bitwise inclusive OR of the operand values.

|	0	1
0	0	1
1	1	1

- ~ is the unary bitwise complement operator (`~x = (-x -1)`).

III.4.2 Coercion and Conversions

- You can perform arithmetic operations and comparisons between any two numbers. If the operands' types differ, coercion applies: Python converts the operand with the smaller type to the larger type.

- The types, in order from smallest to largest, are integers, floating-point numbers, and complex numbers.

- You can also perform an explicit conversion by passing a numeric argument to any of the built-ins fonctions: **`int()`**, **`float()`**, and **`complex()`**.

Note: `int()` function drops it's argument's fractional part, if any.

Note: converting from a complex number to any other numeric type drops the imaginary part.

- Each built-in type can also take a string argument with the syntax of an appropriate numeric literal.

- `int()` an also be called with two arguments: the first one a string to convert, and the second one the radix, an integer between 2 and 36 to use as the base for the conversion.

```
>>> n=12
>>> f=23.75
>>> n=int(f)
>>> n
23
>>> n=int('+45')
>>> n
45
>>> n=int('1010', 2)
>>> n
10
```

III.4.3 The complex type

- A complex number has a real and an imaginary component, both represented by a float attribute.

- An imaginary number is a multiple of the square root of minus one, which is denoted by a **`j`**.

- To represent a complex number you use the syntax:

  `complex = real` **`+`** *`imaginary`***`j`**

- You can access the different parts of the complex number by using the **`real`** and **`imag`** (read only) attributes of the complex object.

```
>>> c = 3.0 + 1.2j
>>> c
(3+1.2j)
>>> print(c.real, c.imag)
3.0 1.2
```


III.5 Simple types are objects

- In Python the simple data types are full-fledged objects, with their own methods and classes.

- If we ask the Python interpreter for information about the `int` object by using the built-in `help`()function we obtain the following result:

```
>>> help(int)
Help on class int in module __builtin__:

class int(object)
 |  int(x[, base]) -> integer
 |
 |  Convert a string or number to an integer, if possible.  A floating point
 |  argument will be truncated towards zero (this does not include a string
 |  representation of a floating point number!)  When converting a string, use
 |  the optional base.  It is an error to supply a base when converting a
 |  non-string. If the argument is outside the integer range a long object
 |  will be returned instead.
 |
 |  Methods defined here:
 |  |  __abs__(...)
 |  |      x.__abs__() <==> abs(x)
 |  |  __add__(...)
 |  |      x.__add__(y) <==> x+y
...
```

- The `help()` function demonstrates that `int` is a class that inherit from the class `object`. The class `int` offers a constructor and several methods `__abs__()`, `__add__()`,…

- A **constructor** is just a special method that creates an instance (an object) of a particular class.

- In Python, a constructor has the same name as the class it creates.

- Following are examples of the use of the `int` class constructor

```
>>> int()
0                       # Create an integer with the default value 0
>>> int(100)            # Create an integer with the value of 100
100
>>> int("100", 10)      # Create an integer with the value of 100 in base 10
100
>>> int("100", 8)       # Create an integer with the value of 100 in base 8
64
```

Note: if you omit the parentheses from the constructor call, you assign the actual class name to the variable, effectively creating an alias to the original class!

- If you want to know more about the help facilities within Python, just enter `help()` at a command prompt in the Python interpreter to access the interactive help utility:

```
>>> help()

Welcome to Python 3.6s help utility!
….

help> int
describes the int class
```

III.5.1 Constructors of simple types

- For the other simple built-in data types, Python does a lot of the work for you. But the constructors are still there -- their names are identical to the name of the relevant data type -- and if you prefer, you can use them directly, as shown below:

```
>>> b = bool(True)
>>> i = int(100)
>>> f = float(100.1)
>>> c = complex(3.0, 1.2)
>>> print(b, i, f, c)
True 100 100.1 (3+1.2j)
```


IV THE CONTAINER TYPES

- When your program needs to handle several objects at once, you can utilize the Python container classes:

  `str`
  `bytes`
  `tuple`
  `list`
  `set`
  `frozenset`
  `dict`
  and also: **`array`**, **`bytearray`**, **`deque`**, …

- These container types provide different capabilities: the first four types are ***sequences*** , `set` and `frozenset` do represent unordered collections of unique elements, a `dict`, is a ***map***.

- A sequence type is just a sequence of objects. All the sequence types' support accessing the objects in the sequence given their order.

- In contrast, *map* and *set* containers hold objects for which the order is not important. A value is extracted from a *map* by providing a ***key*** that locates the ***value*** of interest. When using a *set* the values are retrieved with the help of an *iterator*.

- Another difference between the container types results from the nature of the data they hold: the following four container types are ***immutable***:

  `tuple, str, bytes, frozenset`

- This means that when you create one of these container types, the data being stored can't be changed. If you need to change the data for some reason, you need to create a new container to hold the new data.

- The last three container types (`list`, `set`, and `dict`) are all ***mutable*** containers, so any data they hold can be changed as needed.

- While mutable containers are very flexible, their dynamic nature can result in a performance hit. For example, the `tuple` type, while less flexible because it's immutable, is generally faster than the `list` type when used in identical situations.

- These container classes provide a great deal of power and are often central to most Python programs.

IV.1 Common Sequence Operations

- There are certain things you can do with all sequence types. These operations include ***indexing***, ***slicing***, ***adding***, ***multiplying***, and ***checking for membership***.

- In addition, Python has built-in functions for finding the ***length*** of a sequence, and for finding its ***largest*** and ***smallest*** elements.

- All sequences also offer a way to iterate (to perform certain actions repeatedly, once per element in the sequence) over their elements.

IV.1.1 Indexing

- All elements in a sequence are numbered from zero and upwards.

- You can access them individually given their position using the **`[]`** operator.

```
>>> l=[12,34,56,67,8,88]
>>> l[2]
56
```

IV.1.2 Slicing

- Just as you use indexing to access individual elements, you can use slicing to access ranges of elements.

```
>>> l=[12,34,56,67,8,88]
>>> l[2:4]
[56, 67]
```

IV.1. 3 Adding Sequences

- Sequences can be concatenated with the addition (plus) operator:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> 'Hello, ' + 'world!'
'Hello, world!'
```

- In general, you can only concatenate two sequences of the same kind.

IV.1.4 Multiplication

- Multiplying a sequence by a number x creates a new sequence where the original sequence is repeated x times:

```
>>> [42] * 10
[42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```


IV.1.5 Membership

- To check whether a value can be found in a sequence, you use the **`in`** operator.

```
>>> l=[12,34,56,67,8,88]
>>> n=34
>>> if n in l:
...     print("n is in l")
...
n is in l
```

Note: the operator `in` is also available to maps and sets

IV.1.6 Length, Minimum, and Maximum

- The built-in functions **`len()`** returns the number of elements a sequence contains, while **`min()`** and **`max()`** return the smallest and largest element of the sequence respectively.

Note: the `len()` built-in is also available to maps and sets

IV.2 The tuple type

- The `tuple` type is like a bag into which you throw everything you might need. The Python `tuple` type can hold different types of objects.

- You can create a `tuple` by simply assigning a sequence of objects, separated by commas (`,`), to a variable or using a constructor:

```
>>> t = (0,1,2,3,4,5,6,7,8,9)
>>> type(t)
<type 'tuple'>
>>> t
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tt = 0,1,2,3,4,5,6,7,8,9
>>> type(tt)
<type 'tuple'>
>>> tt
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tc=tuple((0,1,2,3,4,5,6,7,8,9))
>>> tc
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> et = ()      # An empty tuple
>>> et
()
>>> st = (1,)    # A single item tuple
>>> st
(1,)
```

- In the above example, the first method creates a `tuple` that contains the sequence of integers from 0 to 9. The second method does the same, but this time the enclosing parentheses are omitted.

- When creating a `tuple`, the enclosing parentheses are often optional, but sometimes required, depending on the context. The *`tc`* `tuple`, is created thanks to the actual class constructor. The important point here is that the constructor form takes only one argument, so you have to enclose the object sequence within parentheses.

- The last two constructor calls demonstrate how to create an empty tuple (*`et`*) by placing nothing within the enclosing parentheses, and a tuple with only one item (*`st`*) by placing a single comma after the only item in the sequence.

- Creating a single item tuple has the complication of requiring a comma following the single item. This is necessary to differentiate the single-item tuple from a method call.

IV.2.1 Accessing elements in the tuple

- Most of the container types in Python, including the `tuple`, allow you to access items easily from the collection using the ***square bracket*** operator (**`[]`**).

- Python uses zero ordering: the items in a collection are numbered starting with zero.

- Furthermore, you can select one item or multiple sequential items using what is commonly known as ***slicing***.

- The term *slicing* refers to the way in which you are slicing items from the sequence. Slicing is an extremely powerful concept. The general form is *`tuple`*`[start:end:step]`, where start and end are the starting and ending indexes (the corresponding item is not included in the resulting tuple), respectively, and step is the number of items to step over when slicing. So, *`t[2:7]`* slices the third through the seventh items from the tuple (indexes 2,3,4,5 and 6), while *`t[2:7:2]`* slices every the item with indexes 2, 4 and 6.

- You can also use negative values for the ending index, which counts back from the end of the sequence. Another useful feature is that errors -- such as exceeding the length of the sequence -- are generally handled in a graceful manner. You can also choose to omit one or more of the three values used in slicing. For example, I left off the ending index of the slice *`t[2::2]`*.

```
>>> t = (0,1,2,3,4,5,6,7,8,9)
>>> t[2]                    # select a single item, the one at position 2
2
>>> type(t[2])
<type 'int'>
>>> t[0], t[1], t[9]        # create a new tuple whose values are the first, second,
(0, 1, 9)              # 10th values from the initial tuple.

>>> t[2:7]                  # Slice out five elements from the tuple
(2, 3, 4, 5, 6)
>>> type(t[2:7])
<type 'tuple'>
>>> t[2:7:2]                # Slice out three elements from the tuple
(2, 4, 6)
>>> t[2::2]                 # Slice out 4 elements from the tuple (⇔ t[2:10:2])
(2, 4, 6, 8)
>>> t[::]                   # copy of all elements ⇔ t[:]
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> t[::-1]                 # all elements but in the opposite order
(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
>>> t[-4:]                  # last four elements
(6, 7, 8, 9)
```


- A `tuple` is actually a heterogeneous container, as demonstrated in the following examples:

```
>>> t = (0,1,"two",3.0, "four", (5, 6))
>>> t
(0, 1, 'two', 3.0, 'four', (5, 6))
>>> t[1:4]
(1, 'two', 3.0)
>>> type(t[2])
<type 'str'>
>>> type(t[3])
<type 'float'>
>>> type(t[5])
<type 'tuple'>
```

Note: *`t[0] = 10`* is forbidden because *`t`* is a tuple, and a tuple is immutable.

- You can obtain the number of items in a `tuple` using the **`len()`** function.

- Accessing an item from within a nested tuple is also straightforward: you select the nested tuple, and then you access the item of interest from it.

- If you need to create a new `tuple` that contains a subset of the items in an existing `tuple`, the simplest technique is to use the relevant slices and add the subsets together as necessary:

```
>>> tn = t[1:3] + t[3:6]       # Add two tuples
>>> tn
(1, 'two', 3.0, 'four', (5, 6))
>>> tn = t[1:3] + t[3:6] + (7,8,9,"ten")
>>> tn
(1, 'two', 3.0, 'four', (5, 6), 7, 8, 9, 'ten')
>>> len(tn)                    # Find out how many items are in the tuple
9
>>> tn[4][0]                   # Access a nested tuple
5
```


- You can create a `tuple` from a set of existing variables in a process called ***packing***. The opposite is also true, where the values in a `tuple` are assigned to variables. This latter process is known as ***unpacking*** and is an extremely powerful technique used in a number of situations, including when you want to return multiple values from a function.

Note: when you unpack a `tuple` you must have a variable for every item in the `tuple`.

```
>>> i = 1
>>> s = "two"
>>> f = 3.0
>>> t = (i, s, f)          # Pack the variables into a tuple
>>> t
(1, 'two', 3.0)
>>> ii, ss, ff = t         # Unpack the tuple into the named variables
>>> ii
1
>>> ii, ff = t             # Not enough variables to unpack the three element tuple
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: too many values to unpack
```

- However, there are cases where one might only be interested in the first few items of the iterable.
  For example, if you have a list of 5 items

```
>>> values = [23,45,67,78,66]
```

and are only interested in the first two, you would need to do either:

```
>>> a, b, _, _, _ = values
```

or use the ****variable*** syntax to avoid having to write out as many variables than items in the iterable.

```
>>> a, b, *others = values
>> others
[67,78,66]
```

- This can be used for example to access the first two lines and the last line in a file:

```
>>> f = open('data.txt')
>>> first, second, *rest, last = f.readlines()
>>> f.close()
```


IV.3 The string type

IV.3.1 Overview

- Unlike many other programming languages, the Python language does not include a special data type to handle a single character, such as "a" or "z." Python uses a class designed especially for holding sequences of characters, the **`str`** class.

- A string is an immutable sequence of characters.

- To create string object in Python you simply place the desired text inside a pair of quotation marks (as usually, you can also create a string using constructors).

- There are two types of quotations you can use: single quotation marks (`'`) and double quotation marks (`"`). You can use either type of quotation mark to indicate a string in Python, as long as you're consistent. If you start a string with a single quotation mark, you must end with a single quotation mark, and vice versa.

- You can mix single and double quotation marks when creating a string, as long as the string uses the same type of quotation mark at the beginning and end.

- If a string is too long for a single line, you can wrap the string using the Python continuation character: the backslash (****). Internally, the newline character is ignored when creating the string, as is shown when the string is printed.

```
>>> sr="Discover Python"
>>> type(sr)
<type 'str'>
>>> sr='Discover Python'
>>> type(sr)
<type 'str'>
>>> sr="Discover Python: It's Wonderful!"
>>> sr='Discover Python"
  File "<stdin>", line 1
    sr='Discover Python"
                       ^
SyntaxError: EOL while scanning single-quoted string
>>> sr="Discover Python: \
… It's Wonderful!"
>>> sr
Discover Python: It's Wonderful!
```

- To insert a new line into a string, you can use the newline control character (**\n**); to indicate that a tab should be inserted, you can use the tab control character (**\t**) .

```
>>> sr="Discover Python:\n\
… It's Wonderful!"
>>> sr
Discover Python:
It's Wonderful!
```

- A ***raw*** string, is a string in which the control characters are not applied. You create a raw string by preceding the starting quotation mark by an **r** (r is short for raw) .

```
>>> sr=r"Discover Python:\nIt's Wonderful!"
>>> sr
Discover Python:\nIt's Wonderful!
```

- Python provides another way to create long strings that preserves the formatting you use when creating the string. This technique uses three double quotation marks (""") (or three single quotation marks: ''') to begin and end the long string. Within in the string, you can use as many single and double quotation marks as you like.

```
>>> passage = """
...   \tWhen using the Python programming language, one must proceed
...   with caution. This is because Python is so easy to use, and
...   can be so much fun. Failure to follow this warning may lead
...   to shouts of "WooHoo" or "Yowza".
... """
>>> passage
      When using the Python programming language, one must proceed
with caution. This is because Python is so easy to use, and
can be so much fun. Failure to follow this warning may lead
to shouts of "WooHoo" or "Yowza".
```

IV.3.2 The string as an object

- A string object is an instance of the **`str`** class. As you saw previously, the Python interpreter includes a built-in help facility, which can provide information on the `str` class.

```
>>> help(str)

Help on class str in module __builtin__:

class str(basestring)
|  str(object) -> string
…
```

- The strings created using the single, double, or triple quotation mark syntax are still string objects. But you can also explicitly create a string object by using the `str` class constructor, as shown below. The constructor can take a simple built-in numerical type or character data. Either way, the input is changed into a new string object.

```
>>> str("Discover python")
'Discover python'
>>> str(12345)
'12345'
>>> str(123.45)
'123.45'
```

- You can also create a new string by adding other strings together, either using the **+** operator or by just sticking strings together using the appropriate quotes.

- If you need to repeat a small string to create a bigger string, you can use the ***** operator, which multiplies a string out a set number of times.

```
>>> "Wow," + " that " + "was awesome."
'Wow, that was awesome.'
>>> "Wow,"" that ""was Awesome"
'Wow, that was Awesome'
>>> "Wow! "*5
'Wow! Wow! Wow! Wow! Wow! '
>>> sr = str("Hello ")
>>> id(sr)
5560608
>>> sr += "World"
>>> sr
'Hello World'
>>> id(sr)# a new string object was created to hold the result of adding text to the
3708752    # original string, the id is different, this demonstrate that a string is
          # immutable !
```


- The `str` class contains a large number of useful methods for manipulating strings. Let's look at four functions that are useful in their own right and demonstrate the utility of the rest of the `str` class methods.

- The following example demonstrates the **`upper()`**, **`lower()`**, **`split()`**, and **`join()`** methods.

- The first two methods (**`upper()`** and **`lower()`**) are easy to understand. They simply convert the string to all uppercase or all lowercase letters, respectively.

```
>>> sr = "Discover Python!"
>>> sr.upper()
'DISCOVER PYTHON!'
>>> sr.lower()
'discover python!'
```

- The **`split()`** method is useful because it splits a string into a sequence of smaller strings, using a token character (or any character in a given sequence of characters) as an indicator of where to chop.

```
>>> sr = "This is a test!"
>>> sr.split()
['This', 'is', 'a', 'test!']
>>> sr = '0:1:2:3:4:5:6:7:8:9'
>>> sr.split(':')
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

- The first `split()` method example splits the string "This is a test!" using the default token, which is any whitespace character (a space, a tab character, and newline characters).

- The second `split()` method demonstrates using a different token character (in this case, a colon) to split a string into a sequence of strings.

- The **`join()`** method, is the opposite of the `split()` method. This method is used to make a big string from a sequence of smaller strings.

```
>>> sr=":"
>>> tp = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')
>>> sr.join(tp)
'0:1:2:3:4:5:6:7:8:9'
```


IV.3.3 The string as a container for characters

- At the beginning of this chapter, it is mentioned that a Python string is an immutable sequence of characters. Like a `tuple`, a string offers ways to grab a single element, add different elements together, slice out several elements, and even add together different slices.

- One very useful feature of slicing is that slicing too much, going before the start or past the end, doesn't throw an exception but simply defaults to the start or end of the sequence, as appropriate. In contrast, if you try to access a single element with an index outside the allowed range, you get an exception. This behavior demonstrates why the `len()` function is important.

```
>>> sr="0123456789"
>>> sr[0]
'0'
>>> sr[1] + sr[0]
'10'
>>> sr[4:8]     # Returns the elements four through seven, inclusive
'4567'
>>> sr[:-1]     # Returns all elements but the last one
'012345678'
>>> sr[1:12]    # Slice more than you can chew, no problem
'123456789'
>>> sr[:-20]    # Go before the start?
''
>>> sr[12:]     # Go past the end?
''
>>> sr[0] + sr[1:5] + sr[5:9] + sr[9]
'0123456789'
>>> sr[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>> len(sr)    # Sequences have common functions, like len()
10
```


IV.3.4 String formatting

IV.3.4.1 The formatting operator

Note: using the newer ***formatted string literals*** (see § IV.3.4.3) or the **`format`** method **`()`** (see § IV.3.4.2) do, usually, provide more powerful, flexible and extensible approaches to formatting text.

- String formatting can be done with the string **formatting operator**, the percent (`%`) sign.

- To the left of the formatting operator you place a string (the *format string*), and to the right of it you place the value you want to format.

- The right operand can either be a single value, a tuple of values (if you want to format more than one value), or a dictionary.

```
>>> format = "Nb in decimal is %d, and in hexa %x"
>>> nb=34
>>> values = (nb, nb)
>>> format % values
Nb in decimal is 34, and in hexa 22
```

- The formatting operator is the Python equivalent of the C library function `sprintf`.
  The format string is composed of text and/or "conversion specifiers". A "conversion specifiers" begin with the character **`%`** followed by a "conversion character" that specifies the format to use to display the corresponding argument.

Note: To actually include a percent sign in the format string, you must write `%%`.

- If the right operand is a tuple, each of its elements is formatted separately, and you need a conversion specifier for each of the values.

- Following is a description of the main conversion characters

Character	Meaning
`d` or **`i`**	to represent an integer value, in decimal
`o`	to represent an integer value, in octal
`x`	to represent an integer value, in hexadecimal (lowercase)
`X`	to represent an integer value, in hexadecimal (uppercase)
`f` or **`F`**	to represent a float value
`e` or **`E`**	to represent a float value in "scientific notation"
`g` or **`G`**	If the exponent is less than -4 or greater than or equal to the precision, then convert floating-point number as for `%e` or `%E`. Otherwise convert as for `%f`. Trailing zeroes and a trailing decimal point are omitted.
`c`	to represent a character (integer or single character string are accepted)
`s`	to represent a string
`r`	to represent a string (using `repr()`)

- Some optional characters can be inserted between `%` and the conversion character:
  - **`-`** to left-justify the display (by default it is right-justified)
  - a number, that indicates the minimum amount of characters to use to represent the corresponding value. If this is an `*` (asterisk), the width will be read from the value tuple.
  - a dot **`.`** followed by a number to specify the number of digits to use in the fractional part of a float representation (`f`,`e`,`E`) or the total number of digits to appear on both side of the dot for to `g` conversion.
  - **`+`** to print the sign (+ or -) of the corresponding number.
  - ***`space`*** to add space to the beginning of the number if the first character isn't a sign.
  - **`0`** to pad the numbers on the left with zeroes instead of spaces.

```
>>> format = "Pi with three decimals: %10.3f"
>>> from math import pi
>>> print(format % pi)
Pi with three decimals:      3.142
```

```
>>> format = "Pi : %*.3f Nb : %*d"
>>> from math import pi
>>> nb=123
>>> print(format % (10, pi, 8, nb))
Pi :      3.142 Nb :      123
```

String Formatting with Dictionaries

- As stated earlier, the right operand of the formatting operator can be a dictionary (with only strings as keys).

- In that situation, after the `%` character in each conversion specifier, you add a key (enclosed in parentheses), which is followed by the other specifier elements (except for the added string key, the conversion specifiers work as before).

- According to the key being used, the formatting operator will select an entry in the dictionary.

```
>>> phone={'marc' : 2234, 'eliane' : 3345, 'michael' : 3334}
>>> format = "Marc phone: %(marc)6s Michael phone:
%(michael)6s"
>>> print(format % phone)
Marc phone:   2234  Michael phone:   3334
```


IV.3.4.2 Formatting string using the format() method.

- The `str format()` method was added in Python 2.6 and 3.0 at the same time (there is also a global built-in function, `format()`).

- The general form of this method looks like this:

  template-string.**format(***p0*, *p1*, ..., *k0=v0*, *k1=v1*, ...**)**

- The *template* (or *format*) *string* is a string which contains one or more format codes (fields to be replaced) embedded in "normal" text. The "fields to be replaced" are surrounded by curly braces **{}**.

- The curly braces and the "code" inside will be substituted with a formatted value from one of the arguments, according to the rules which we will specify soon.
  Anything else, which is not contained in curly braces will be literally printed, i.e. without any changes.

Note: if a brace character has to be printed, it has to be escaped by doubling it: `{{` and `}}`.

The arguments

- There are two kinds of arguments for the `format()` method: the list of arguments starts with zero or more positional arguments (*p0*, *p1*, ...), and it may be followed by zero or more keyword arguments of the form *name=value*: *k0=v0*, *k1=v1*, ...

- In the template string, the positional parameter of the `format()` method can be accessed by placing the index of the parameter after the opening brace, e.g. `{0}` accesses the first parameter, `{1}` the second one and so on. The index inside of the curly braces can be followed by a colon and a format string, for instance `{0:5d}`.

- If the positional parameters are used in the order in which they are written, the argument index inside of the braces can be omitted, so '*{} {} {}*' corresponds to '*{0} {1} {2}*'. But they are needed, if you want to access them in different orders: '*{2} {1} {0}*',

```
>>> artNb=234
>>> price=34.567
>>> "Art: {0:5d}, Price: {1:.2f}".format(artNb, price)
Art:   234, Price: 34.57
>>> "Art: {:5d}, Price: {:.2f}".format(artNb, price)
Art:   234, Price: 34.57
```

- Arguments can be used more than once:

```
>>> nb=56
>>> print("Nb: {0:d} Nb in hexa: {0:x}".format(nb))
Nb: 56 Nb in hexa: 38
```

- In the following example we demonstrate how keyword parameters can be used with the `format()` method:

```
>>> artNb=234
>>> "Art: {a:5d}, Price: {p:.2f}".format(a=artNb, p=34.567)
Art:   234, Price: 34.57
```

- This case can be expressed with a dictionary as well, as we can see in the following code:

```
>>> data={"a":234, "p":34.567}
>>> "Art: {a:5d}, Price: {p:.2f}".format(**data)
Art:   234, Price: 34.57
```

- The double "*****" in front of *data* turns *data* automatically into the form *a=234, p=34.567*.

The format placeholder

- The general syntax for a format placeholder is

  {[position or name]: [[fill]align][sign][#][0][width][,][.precision][type]}

- If a valid ***align*** value (alignment options) is specified, it can be preceded by a ***fill*** character that can be any character and defaults to a space if omitted.
  The meaning of the various alignment options is as follows

Align	Meaning
<	The field will be left-aligned within the available space. This is the default for strings.
>	The field will be right-aligned within the available space. This is the default for numbers.
^	Forces the field to be centered within the available space.
=	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form "+000000120". This alignment option is only valid for numeric types.

- We can modify the formatting with the ***sign*** option, which is only valid for number types:

Sign	Meaning
+	indicates that a sign should be used for both positive and negative numbers.
–	indicates that a sign should be used only for negative numbers (the default).
space	indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

Meaning of ***0***, ***#*** and ***,*** :

0	If the width field is preceded by a zero (0) character, sign-aware zero-padding for numeric types will be enabled.
	>>> x = 378
	>>> "The value is {:06d}".format(x)
	'The value is 000378'
	>>> x = -378
	>>> "The value is {:06d}".format(x)
	'The value is -00378'

,	This option signals the use of a comma for a thousands separator.
	>>> x=789667324245
	>>> "The value is {:,}".format(x)
	'The value is 789,667,324,245'

#	Used with o, x or X specifiers the value is preceded with 0, 0o, 0O, 0x or 0X respectively.

- ***width*** is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content (the alignment option has no meaning in this case).

- The ***precision*** is a decimal number indicating how many digits should be displayed after the decimal point for a floating point value formatted with 'f' and 'F', or before and after the decimal point for a floating point value formatted with 'g' or 'G'. For non-number types the field indicates the maximum field size - in other words, how many characters will be used from the field content.

- The ***type*** determines how the data should be presented.

Type	Meaning
s	String format. This is the default type for strings and may be omitted.
b	Binary format. Outputs the number in base 2.
c	Character. Converts the integer to the corresponding unicode character before printing.
d	Decimal Integer. Outputs the number in base 10.
o	Octal format. Outputs the number in base 8.
x	Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.
X	Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9.
n	Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters.
e	Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent. The default precision is 6.
E	Exponent notation. Same as 'e' except it uses an upper case 'E' as the separator character.
f	Fixed point. Displays the number as a fixed-point number. The default precision is 6.
F	Fixed point. Same as 'f', but converts nan to NAN and inf to INF.
g	General format. For a given precision p >= 1, this rounds the number to p significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. The default precision is 6. Positive and negative infinity, positive and negative zero, and nans, are formatted as inf, -inf, 0, -0 and nan respectively, regardless of the precision.
G	General format. Same as 'g' except switches to 'E' if the number gets too large. The representations of infinity and NaN are uppercased, too.
n	Number. This is the same as 'g', except that it uses the current locale setting to insert the appropriate number separator characters.
%	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.
None	Similar to 'g', except with at least one digit past the decimal point and a default precision of 12. This is intended to match str(), except you can add the other format modifiers.

- The `format()` method calls by default the special **`__format__()`** method of an object for its representation. If you just want to render the output of `str()` or `repr()` you can use the **`!s`** or **`!r`** conversion flags.

```
class Data(object):
     def __str__(self): return 'str'
     def __repr__(self): return 'repr'
     def __format__(self,fmt_spec): return 'format'

print('{0} {0!s} {0!r}'.format(Data()))
# output: format str repr
```

IV.3.4.3 Literal String Interpolation (Python 3.6)

- A string literal with '**f**' or '**F**' in its prefix is a formatted string literal (or ***f-string*** or ***f-literal***) a feature introduced in Python 3.6.

- F-strings provide a concise, readable way to include the value of Python expressions inside strings, using a minimal syntax.

```
>>> artNb=234
>>> price=34.567
>>> f"Art: {artNb:5d}, Price: {price:.2f}"
Art:   234, Price: 34.57
```

- Expressions appear within curly braces (**{}**).

Note: any doubled braces '{{' or '}}' inside literal portions of an f-string are replaced by the corresponding single brace.

- Neither backslashes nor comments are allowed inside an expression.

- Leading and trailing whitespaces in expressions are ignored.

- Following each expression, an optional ***type conversion*** may be specified. The allowed conversions are '**!s**' (calls `str()` on the expression) , '**!r**' (calls `repr()`), or '**!a**' (calls `ascii()`).

```
>>> text="déjà"
>>> str(text)
déjà
>>> repr(text)
'déjà'
>>> ascii(text)
'd\xe9j\xe0'
>>> f'text in ascii: {text!a}'
text in ascii: 'd\xe9j\xe0'
>>> # <=>
>>> f'text in ascii: {ascii(text)}'
text in ascii: 'd\xe9j\xe0'
```

- Similar to `str.format()`, optional format specifiers may be included inside the f-string, separated from the expression (or the type conversion, if specified) by a colon.

```
>>> import datetime
>>> name = 'Marco'
>>> age=22
>>> anniversary = datetime.date(1995, 9, 11)
>>>  f'Name: {name}, age next year: {age+1}, anniversary: {anniversary:%A, %B %d, %Y}.'
'Name: Marco, age next year: 23, anniversary: Monday, September 11, 1995.'
>>> # this is equivalent to:
>>>'Name: {name}, age next year: {age}, anniversary: {anniversary:%A, %B %d, %Y}.'.format(name=name, age=age+1, anniversary=anniversary)
'Name: Marco, age next year: 23, anniversary: Monday, September 11, 1995.'
>>> # and also equivalent to:
>>> 'Name is {}, age next year: {}, anniversary: {:%A, %B %d, %Y}.'.format(name, age+1, anniversary)
'Name: Marco, age next year: 23, anniversary: Monday, September 11, 1995.'
```

- Format specifiers may also contain evaluated expressions:

```
>>> import decimal
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal('12.34567')
>>> f'result: {value:{width}.{precision}}'
'result:      12.35'
```

Note: the '`f`' may be combined with '`r`', but not with '`b`' : raw formatted strings are possible, but formatted bytes literals are not.

IV.3.4.4 Other string methods for formatting

- The string class contains further methods, which can be used for formatting purposes as well: **`ljust()`**, **`rjust()`**, **`center()`** and **`zfill()`**.

IV.3.5 Other string methods and functions

- A string being a sequence, all the standard sequence operations (indexing, slicing, multiplication, membership, length, minimum, and maximum) work with strings.

- **`len(`*`str`*`)`** : returns the length of a string.

- **`min(`*`str`*`)`** and **`max(`*`str`*`)`** return the smallest and largest character of the string respectively.

Note: These are functions, not methods. You must call these functions with a sequence argument (here a string).

```
>>> text="abc 12"
>>> len(text)
6
>>> max(text)
'c'
>>> min(text)
' '
>>> text="abc" * 3
>>> text
abcabcabc
>>> if 'd' in text:
...     print("d is present")
... else:
...     print("d is absent")
...
d is absent
>>> text[0]
a
>>> text[0:3]
abc
```


find()

- The **find()** method finds a substring within a larger string. It returns the leftmost index where the substring is found.

- If it is not found, –1 is returned.

```
>>> text="The string method find does not return a Boolean"
>>> text.find("method")
11
>>> text.find("The")
0
```

- You may also supply a starting point for your search and, optionally, also an ending point:

```
>>> text="The string method of the string class"
>>> text.find("string")
4
>>> text.find("string",10)
25
>>> text.find("string",10,20)
-1
```

replace()

- The **replace()** method returns a string where all the occurrences of one string have been replaced by another:

```
>>> text="The string method of the string class"
>>> text.replace("string", "int")
'The int method of the int class'
```

strip()

- The strip method returns a string where whitespace on the beginning and at the end (but not internally) has been stripped (removed):

```
>>> text="   Hello    World   "
>>> text.strip()
'Hello    World'
```

- You can also specify which characters are to be stripped, by listing them all in a string parameter:

```
>>> text=" --- *** Hello    World  *** --- "
>>> text.strip(" - *")
'Hello    World'
```


`translate()`

- Similar to `replace()`, **`translate()`** replaces parts of a string, but unlike `replace()`,`translate()` only works with single characters.

- Before you can use translate, you must make a ***translation table***.

- This translation table is a full listing of which characters should be replaced by which.

- Because this table (which is actually just a string) has 256 entries, you won't write it out yourself: you'll use the function **`maketrans()`** from the `string` module.

- The `maketrans()` function takes two arguments: two strings of equal length, indicating that each character in the first string should be replaced by the character in the same position in the second string.

```
>>> from string import maketrans
>>> table = maketrans('aeiouy', 'AEIOUY')
>>> text="This is a test"
>>> text.translate(table)
ThIs Is A tEst
```

- An optional second argument can be supplied to `translate()`, specifying letters that should be deleted.

```
>>> from string import maketrans
>>> table = maketrans('aeiouy', 'AEIOUY')
>>> text="This is a test"
>>> text.translate(table, 'tT')
hIs Is A Es
```

IV.3.6 String conversions

- The built-in function **`repr()`** evaluates the argument it receives and converts the resulting object into a string according to rules specific to its type.

- If the object is a string, a number, `None`, or a `tuple`, `list` or `dictionary` containing only objects whose type is one of these, the resulting string is a valid Python expression. This string could be passed to the built-in function **`eval()`** to yield an expression with the same value.

- The built-in function **`str()`** performs a similar but more user-friendly conversion.

Note: Python 2 do also include ***string conversion***: an expression list enclosed in back quotes (`` `expression` ``). It works in a way similar to the built-in function **`repr()`**.

IV.3.7 Other methods

capitalize()	Returns a copy of the string in which the first character is capitalized
center(width[, fillchar])	Returns a string of length max(len(string), width) in which a copy of string is centered, padded with fillchar (default is space)
count(sub[, start[, end]])	Counts the occurrences of the substring sub, optionally restricting the search to string[start:end]
endswith(suffix[, start[, end]])	Checks whether string ends with suffix, optionally restricting the matching with the given indices start and end
expandtabs([tabsize])	Returns a copy of the string in which tab characters have been expanded using spaces, optionally using the given tabsize (default 8)
index(sub[, start[, end]])	Returns the first index where the substring sub is found, or raises a ValueError if no such index exists, optionally restricting the search to string[start:end]
isalnum() **isalpha()** **isdigit()** **islower()** **isspace()** **isupper()**	Work as expected …
istitle()	Checks whether all the case-based characters in the string following non-casebased letters are uppercase and all other case-based characters are lowercase
lstrip([chars])	Returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters)
rfind(sub[, start[, end]])	Similar to find but start the search from the end of the string
rindex(sub[, start[, end]])	Similar to index but start the search from the end of the string
rstrip([chars])	Returns a copy of the string in which all chars have been stripped from the end of the string (default whitespace characters)
rsplit([sep[, maxsplit]])	Same as split, but when using maxsplit, counts from right to left
splitlines([keepends])	Returns a list with all the lines in string, optionally including the line breaks (if keepends is supplied and is true)
startswith(prefix[, start[, end]])	Checks whether string starts with prefix, optionally restricting the matching with the given indices start and end
swapcase()	Returns a copy of the string in which all the case-based characters have had their case swapped
title()	Returns a copy of the string in which all the words are capitalized

IV.4 The bytes type

IV.4.1 Overview

- A `bytes` object is immutable; if you need to change individual bytes you can convert the bytes object into a **`bytearray`** object using the built-in `bytearray()` function.

- All the methods and operations you can do on a `bytes` object, you can do on a `bytearray` object too. The one difference is that, with the `bytearray` object, you can assign individual bytes using index notation.

- The items of a bytes object are integers. Specifically, integers between 0–255.

- In Python, a byte string is represented by a **`b`**, followed by the byte string's ASCII representation.

- Each byte within the byte literal can be an ASCII character or an encoded hexadecimal number from \x00 to \xff (0–255).

- To define a bytes object, use the **`b""`** *byte literal* syntax or the **`bytes()`** built-in function.

- `bytes` objects have a **`decode()`** method that takes a character encoding and returns a string, and strings have an **`encode()`** method that takes a character encoding and returns a bytes.

IV.4.2 bytes vs str

- The only thing that a computer can store is bytes.
  To store anything in a computer, you must first ***encode*** it, i.e. convert it to bytes.
  For example:

    - If you want to store music, you must first encode it using MP3, WAV, etc.
    - If you want to store a picture, you must first encode it using PNG, JPEG, etc.
    - If you want to store text, you must first encode it using ASCII, UTF-8, etc.

- MP3, WAV, PNG, JPEG, ASCII and UTF-8 are examples of **encodings**. An encoding is a format to represent audio, images, text, etc in terms of bytes.

- Historically, programmers and programming languages have tended to explicitly or implicitly assume that a byte sequence and an ASCII string were the same thing. Python 3 decided to explicitly break this assumption.

- In Python, a **byte string** is just that: a sequence of bytes. It isn't human-readable (as mentioned above, under the hood, everything must be converted to a byte string before it can be stored in a computer).

- On the other hand, a **character string**, often just called a "string", is a sequence of characters. It is human-readable.

- A character string can't be directly stored in a computer, it has to be encoded first (converted into a byte string). There are multiple encodings through which a character string can be converted into a byte string, such as ASCII and UTF-8.

```
'I am a string'.encode('ASCII')
```

- The above Python code will encode the string *'I am a string'* using the encoding *ASCII*. The result of the above code will be a byte string. If you print it, Python will represent it as *b'I am a string'*. Remember, however, that byte strings aren't human-readable, it's just that Python decodes them from ASCII when you print them.

- A byte string can be decoded back into a character string, if you know the encoding that was used to encode it.

```
b'I am a string'.decode('ASCII')
```

- The above code will return the original string *'I am a string'*.

- Encoding and decoding are inverse operations. Everything must be encoded before it can be written to disk, and it must be decoded before it can be read by a human.

IV.5 The list type

IV.5.1 Overview

- A **`list`** is a mutable sequence type.

- You can use a `list` like a bag to hold different types of data that can be changed at will. You can use the list like an array to hold data in an organized manner. And you can use the list as a queue or as a stack.

- A Python `list` is very similar to a `tuple`. One difference, however, is that you can create a list using square brackets (**`[]`**). The following example shows how to create a simple list containing the integers from 0 to 9, inclusive, as well as how to create an empty list and a list containing a single item.

```
>>> l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> type(l)
<type 'list'>
>>> el = []          # Create an empty list
>>> len(el)
0
>>> sl = [1]         # Create a single item list
>>> len(sl)
1
>>> sl = [1,]   # Also creates a single item list (the coma is optional)
>>> len(sl)
1
```

- You can create a list using a constructor, in addition to the shorthand notation of using the square brackets.

- The `list` class provides two different constructors. One takes no arguments, and the other takes a sequence class. This provides a great deal of flexibility because you can easily convert an existing sequence like a `tuple` or a `string` into a `list`, as shown in the following example. Notice, however, that you must pass in a sequence (and not just a sequence of objects) or you'll get an error.

- You aren't limited to passing a `sequence` directly into the constructor; you can also pass a variable that holds a `tuple` or a `string` into the `list` constructor.

- As with any sequence type, you can easily find out the number of items in the sequence using the **`len()`** function.

```
>>> l = list()
>>> type(l)
<type 'list'>
>>> len(l)
0
>>> l
[]
>>> l = list((0, 1, 2, 3, 4, 5, 6, 7, 8, 9))# Create a list from a tuple
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(l)
10
>>> l = list([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]) # Create a list from a list
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(l)
10
>>> l = list(0, 1, 2)       # Error: Must pass in a sequence
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list() takes at most 1 argument (3 given)
>>> l = list("0123456789") # Create a list from a string
>>> l
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> type(l)
<type 'list'>
>>> len(l)
10
```

- Like with the other sequences, you can access items from a list one item at a time or by slicing items:

```
>>> l = list([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> l[0]          # Get the first item in the list
0
>>> type(l[0])
<type 'int'>
>>> l[5]          # Get the sixth item in the list
5
>>> l[1:5]        # Get the second through fifth items
[1, 2, 3, 4]
>>> type(l[1:5])
<type 'list'>
>>> l[0::2]       # Get the item 0, 2, 4, …
[0, 2, 4, 6, 8]
>>> l[0], l[1], l[2]
(0, 1, 2)
```

IV.5.2 A mutable sequence

- The main difference between the `list` and the `tuple` is that a list is a ***mutable*** sequence. This means you cannot access AND modify the items of the `list`.

Note: You can only modify items in a list. To add items to a list (not just modify items), you can use the **`append()`** method, as shown below:

```
>>> l = []
>>> l[0] = 0      # The list is empty
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
>>> l.append(0)
>>> l
[0]
>>> l[0] = 1
>>> l
[1]
```

- As the previous example demonstrated, trying to modify a list item that doesn't exist results in an error.

IV.5.3 A heterogeneous mutable list

- A list can hold different types of data, and be modified in different ways.
- Modifying an item within a list is rather simple: You set the item's value appropriately, even to a different type, such as a string or another list.
- Using the slice syntax, without specifying a starting or ending index, you can make a duplicate copy of an existing `list`.
- You can also use the repeat operator (**`*`**), to build up a bigger list from small segments.
- To delete objects from a list, the first method for deleting items is to use the **`del()`** method. This method can be used to delete one item or a range of items. You can also use slicing to delete slices from a list.

```
>>> l=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2] = 2
>>> type(l[2])
<type 'int'>
>>> l[2] = "two"       # Change an element
>>> type(l[2])
<type 'str'>
>>> l
[0, 1, 'two', 3, 4, 5, 6, 7, 8, 9]
>>> l[2] = l[2:5] * 2
>>> l
[0, 1, ['two', 3, 4, 'two', 3, 4], 3, 4, 5, 6, 7, 8, 9]
>>> del(l[2])          # Remove single element
>>> l
[0, 1, 3, 4, 5, 6, 7, 8, 9]
>>> l[1:3] = []        # Remove a slice
>>> l
[0, 4, 5, 6, 7, 8, 9]
>>> bis=l[:]           # Duplicate the list l
>>> print(bis, id(l), id(bis))
[0, 4, 5, 6, 7, 8, 9] 42030280 42013768
```

IV.5.4 List as arrays

- In the previous example, you saw that a list can contain another list as an item. If every item is replaced by a list, the result is a matrix.

- The following example demonstrates how to use a list to hold a two-dimensional (2-D) or three-dimensional (3-D) array.

```
>>> al = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> al
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> al[0][0]           # First element in 2D array
0
>>> al[2][2]           # Last element in 2D array
8
>>> al[1][2]
5
>>> al = [[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
>>> al
[[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
>>> al[0][0][1]
1
>>> len(al)            # Length of outer dimension
2
>>> len(al[0])         # Length of middle dimension
2
>>> len(al[0][0])      # Length of inner dimension
2
```

IV.5.5 Other list operations

- The `list` object has a number of useful methods that can be applied to an existing list. For example, you can **`reverse()`** all the items in a list or **`sort()`** a list. An important point to remember with these operations, however, is that they operate in place, which means they modify the list on which they're called.
- A list can also be used to emulate other data structures in addition to an array. For example, the **`append()`** and **`pop()`** methods operating on a list function as either a First In, First Out (FIFO) data structure; or as a Last In, First Out (LIFO) data structure.
  The `pop()` method supports these capabilities by allowing you to set the item to be popped (removed and returned) from the list. If you pop the first item of a list, you have a queue; whereas, if you pop the last item of a list, you have a stack, as shown below:

```
>>> l=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> id(l)            # This is the object id for our current list
4525432
>>> l.reverse()      # Reverse the list
>>> l
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> id(l)            # The id is the same, modified list in place.
4525432
>>> l.sort()         # Sort the list in numerical order
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> id(l)            # Modified the existing list
4525432
>>> l.index(5)       # Same as l[5]
5
>>> l.count(0)       # How many times does '0' occur in the list
1
>>> l.pop()          # Take off the last item (Stack)
9
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> l.pop(5)         # Take out the fifth element
5
>>> l
[0, 1, 2, 3, 4, 6, 7, 8]
>>> l.pop(0)         # Take the first item off the list (Queue)
0
>>> l
[1, 2, 3, 4, 6, 7, 8]
```


- Another way of getting a sorted copy of a list is using the **`sorted()`** function (note: the `sorted()` function accepts any iterable):

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y = sorted(x)
>>> x
[4, 6, 2, 1, 7, 9]
>>> y
[1, 2, 4, 6, 7, 9]
```

Both the `sort()` method and `sorted()` have a **`key`** parameter to specify a function to be called on each list element prior to making comparisons. The value of the `key` parameter should be a function that takes a single argument and returns a key to use for sorting purposes.

`count(obj)`

- The `count()` method counts the occurrences of an element in a list:

```
>>> x = [[1, 2], 1, 1, [2, 1, [1, 2]]]
>>> x.count(1)
2
```

`extend(seq)`

- The `extend()` method allows you to append several values at once by supplying a sequence of the values you want to append. In other words, the original list has been extended by the other one:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6]
```

`index(obj)`

- The **`index()`** method is used for searching lists to find the index of the first occurrence of a value:

```
>>> knights=['We','are','the','knights','who','say','ni']
>>> knights.index('who')
4
```


`insert(pos, obj)`

- The `insert()` method is used to insert an object into a list:

```
>>> numbers = [1, 2, 3, 5, 6, 7]
>>> numbers.insert(3, 'four')
>>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

`remove(obj)`

- The `remove()` method is used to remove the first occurrence of a value:

```
>>> x = ['to', 'be', 'or', 'not', 'to', 'be']
>>> x.remove('be')
>>> x
['to', 'or', 'not', 'to', 'be']
```


IV.6 Set and Frozenset

IV.6.1 Overview

- A **`set`** (a Python implementation of the sets as they are known from mathematics) contains an unordered collection of unique and immutable objects.

- Though sets can't contain mutable objects, sets are mutable, `frozensets` are immutable.

IV.6.2 Creation

- To create a `set` (a `frozenset`) , we call the built-in **`set()`** function with a an iterable argument.

- We also create sets (since Python 2.6) using curly braces (**`{}`**)instead of the **`set()`** built-in function.

```
>>> s1=set()          # an empty set
>>> print(s1, type(s1))
set() <class 'set'>
>>> s1=frozenset()   # an empty frozenset set
>>> print(s1, type(s1))
frozenset() <class 'frozenset'>
>>> s1={}                    # an empty dict !!
>>> print(s1, type(s1))
{} <class 'dict'>
>>> s1=set([12,23])        # a set with 2 elements in it
>>> print(s1, type(s1))
{12, 23} <class 'set'>
>>> s1=frozenset([12,23])  # a frozenset with 2 elements in it
>>> print(s1, type(s1))
frozenset({12, 23}) <class 'frozenset'>
>>> s1={12,23}        # a set with the 2 same elements in it
>>> print(s1, type(s1), len(s1))
{12, 23} <class 'set'> 2
>>> s1={12,23,12}     # a set with the 2 elements in it (duplicates are not allowed)
>>> print(s1, type(s1), len(s1))
{12, 23} <class 'set'> 2
```

IV.6.3 Instance methods

- **`add(e)`**: adds `e` (`e` has to be immutable), to the set (if `e` is not already contained).
- **`clear()`**:remove all elements from the set.
- **`copy()`**:returns a shallow copy of the set.

```
>>> s1={12,23,12}
>>> s1.add(24)
>>> print(s1)
{24, 12, 23}
>>> # s1.add([25,26]) # not allowed: a list is mutable
>>> s2=s1.copy()       # shallow copy of s1
>>> print(s2, len(s2))
{24, 12, 23} 3
>>> s2.clear()         # all elements are removed
>>> print(s2, len(s2))
set() 0
```

- **`difference(s)`**: returns, as a new `set`, the difference this set and `s` (the **`-`** operator works in a similar way).
- **`difference_update(s)`**: removes all elements of another set from this set (<=> x = x - y)
- **`discard(e)`**: removes `e` from the set (if `e` is not present, `discard()` does nothing)

```
>>> s1={24, 12, 23}
>>> s2={12,23,67,78}
>>> s3=s1-s2
>>> print(s1, s2, s3)
{24, 12, 23} {67, 12, 78, 23} {24}
>>> s3=s1.difference(s2)
>>> print(s1, s2, s3)
{24, 12, 23} {67, 12, 78, 23} {24}
>>> s1.difference_update(s2)
>>> print(s1)
{24}
>>> s2.discard(67)
>>> print(s2)
{12, 78, 23}
```

- **`intersection(s)`**: returns the intersection of the instance set and `s` as a new set (the **`&`** operator works in a similar way).

- **`isdisjoint()`**:returns `True` if two sets have a null intersection.

- **`issubset()`**:returns `True`, if `x` is a subset of `y` (**`<=`**, **`<`** are an abbreviation for "Subset of" and proper "subset of").

- **`issuperset()`**:returns `True`, if `x` is a superset of `y` (**`>=`**, **`>`** are an abbreviation for "Superset of" and proper "superset of").

- **`pop()`**:removes and returns an arbitrary element. The method raises a `KeyError` if the set is empty.

- **`remove(e)`**: same as `discard()`, but raises a `KeyError` if `e` is not present.

- **`union(s)`**: returns the union of the instance set and `s` as a new set (the **`|`** operator works in a similar way).

```
>>> s1={24, 12, 23}
>>> s2={12, 78, 23}
>>> s3=s1|s2
>>> print(s1, s2, s3)
{24, 12, 23} {12, 78, 23} {12, 78, 23, 24}
>>> s3=s1.union(s2)
>>> print(s1, s2, s3)
{24, 12, 23} {12, 78, 23} {12, 78, 23, 24}
>>> s3=s1&s2
>>> print(s1, s2, s3)
{24, 12, 23} {12, 78, 23} {12, 23}
>>> s3=s1.intersection(s2)
>>> print(s1, s2, s3)
{24, 12, 23} {12, 78, 23} {12, 23}
>>> s1={240, 1, 23,666}
>>> r=s1.pop()
>>> print(r, s1)
240 {1, 666, 23}
```


V PROGRAM FLOW

- Python provides three flow control statements:

    - The **`if`** statement, which executes a particular block of statements based on the result of a test expression
    - The **`while`** loop, which executes a block of statements as long as a test expression is true
    - The **`for`** loop, which executes a block of statements a certain number of times

V. 1 The if statement

- The simplest flow control statement is the **`if`** statement, whose basic syntax is demonstrated in the pseudo-code below:

```
if Boolean Expression:
    # Action to take if Boolean Expression evaluates True
else:
    # Action to take if Boolean Expression evaluates False
```

- The `if` statement executes a block of program statements if a *Boolean* expression evaluates *True*. The `if` statement supports an optional **`else`** clause that indicates a block of program statements that should be processed if the *Boolean* expression evaluates *False*.

- Note the termination of the `if` and the `else` statements with the colon character, and the indentation of the code within the `if` and the `else` blocks. As mentioned above, these two characteristics are required in Python for flow control statements.

- In the following example, a simple *`if`*/*`else`* condition tests whether a given number is even or odd and prints out the result.

```
>>> i = 8
>>> if i % 2 != 0 :
...     print("Odd Number")
... else:
...     print("Even Number")
...
Even Number
```

Note: the three dots (...) at the start of all the lines following the if statement are being displayed by the Python interpreter. When you type in the if statement the Python interpreter realizes you have entered a compound statement. As a result, it changes the prompt from three greater than symbols (>>>) to the three dots (...).

- The Boolean expression used in the `if` statement can be composed of multiple sub-expressions that use the different relational operators supported in Python (**`<`**, **`<=`**, **`>`**, **`>=`**, **`==`**, **`!=`**). And the sub-expressions can be combined using the **`and`**, **`or`**, and **`not`** logical operators (see III.3).

- It is possible to cascade several `if` statements: you simply add an extra `if` statement to the `else` clause. The result is an else if statement, which is shortened to **`elif`**, as demonstrated below:

```
>>> i = -8
>>> if i > 0:
...     print("Positive Integer")
... elif i < 0:
...     print("Negative Integer")
... else:
...     print("Zero")
...
Negative Integer
```

Note: You can include as many `elif` as the program needs.

- C and C++, have for years allowed ***conditional expressions***, which are expressions that include an `if` statement. Python, since version 2.5, now also offers this possibility using the syntax:

  `expression1` **`if`** `condition` **`else`** `expression2`

  Python first evaluates *`condition`* (not *`expression1`*); if *`condition`* is true, *`expression1`* is evaluated and its value is returned; otherwise, *`expression2`* is evaluated and its value is returned.

- A traditional `if` statement like the following:

```
if len(l) > 3:
    result=sum(l)
else:
    result=100
```

can also be written in a single expression:

```
result=sum(l) if len(l) > 3 else 100
```


V.2 The while loop

- The second type of flow control statement in Python is the **`while`** loop, which executes a block of program statements while a Boolean expression evaluates *True*.

- The `while` loop supports an optional **`else`** clause containing a block of program statements executed when the expression is `False`. This means the code in the `else` clause is executed once, after the loop is terminated:

```
while Boolean Expression:
    # statements to execute while Boolean Expression is True
else:
    # statements to execute when Boolean Expression is False
```

```
>>> i = 0 ; x = 10
>>> while x > 0:
...     i += 1
...     x -= 1
... else:
...     print(i, x)
...
10 0
```

Note: The above example combines variable initializations and, later, variable modifications on one line thanks to the semicolon separator.

- The `while` loop (like the `for` loop introduced later) supports three additional statements:

```
continue
break
pass
```

- The **`continue`** and **`break`** statements are used inside a while loop to continue with the next pass through the loop or break out of the loop, respectively. Often, these two statements are placed in the body of an `if` statement so that the `continue` or `break` action is triggered by a special condition.

- The `break` statement breaks completely out of the loop, thereby skipping any `else` clause that follows the loop.

- The **`pass`** statement is a do-nothing statement. It's used as a placeholder when a statement is required, but the program logic doesn't require an action.


```
>>> i = 1
>>> while i < 1000:
...     i *= 5
...     if i % 25:
...         continue
...     if not (i % 125):
...         break
...     if not (i % 1000):
...         pass
... else:
...     print(i)
...
>>> print(i)
125
```

- Tracing the logic flow through the program, you see that the first time through the loop, the value of the variable i becomes 5. The first `if` statement evaluates True, because 5 isn't evenly divisible by 25. This starts the second trip through the `while` loop, where the variable i becomes 25. Now the first `if` statement evaluates False because 25 is evenly divisible by 25. The second and third `if` statements also evaluate False, meaning you start the third pass through the loop. This time the variable i becomes 125, and the first `if` statement evaluates False.
  But now the second `if` statement evaluates True because the variable i is evenly divisible by 125 (and the not operator turns the resulting 0 into a Boolean True). This causes the break statement to be executed, breaking out of the loop. The else clause is never executed, so nothing is printed out until you explicitly use a separate `print` function.

V.3 The for loop

- The **`for`** loop in Python is closely tied to the container data types built into the Python programming language.

- The `for` loop is used to iterate through the items in a Python collection, including the Python tuple, string, and list container types discussed in the previous chapters.

- The `for` loop can also be used to access elements from a container type by using the **`range()`** function. In addition, you can use the `range()` function to execute a group of statements a specific number of times within a for loop.

- The `for` loop syntax is demonstrated in the following pseudocode:

```
for item in collection:
    # action to repeat for each item in the collection
else:
    # action to take once we have finished the loop.
```

- The `for` loop involves what is known as an ***iterator***, which is used to move through the collection item by item. The collection can be any Python container type.

- The for loop can also be utilized to iterate over any object that supports the iteration metaphor.

- The following example demonstrates how to use a for loop to iterate through the elements of a `tuple`.

```
>>> t = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> count = 0
>>> for num in t:
...     count += num
... else:
...     print(count)
...
45
>>> count = 0
>>> for num in t:
...     if num % 2:
...         continue
...     count += num
... else:
...     print(count)
...
20
```

- This example first creates a `tuple` named *`t`* that holds the integers 0 to 9, inclusive. The first `for` loop iterates through this `tuple`, cumulating the sum of the numbers in the tuple in the *`count`* variable.

Once the code has iterated over all the elements in the `tuple`, it enters the `else` clause of the `for` loop, which prints out the value of the *`count`* variable.
The second `for` loop also iterates over all elements in the `tuple`. In this case, however, it only cumulates the values of those items in the container that are evenly divisible by 2. This restriction is accomplished by using an appropriate `if` statement with a `continue` statement.

- The next example demonstrates how to use a Python string as the container for a `for` loop.

```
>>> st = "Python Is A Great Programming Language!"
>>> for c in st:
...     print(c, end=" ")
...
P y t h o n   I s   A   G r e a t   P r o g r a m m i n g
L a n g u a g e !
>>> count = 0
>>> for c in st:
...     if c in "aeiou":
...         count += 1
... else:
...     print(count)
...
10
```

- The first for loop iterates over the string "Python Is A Great Programming Language!" and prints each character in the string one at a time. In this case, the print function uses a comma following the variable *c*. This makes the print function print out the character value, followed by a space character instead of a newline character. Without the trailing comma, the characters would all be printed on separate lines.

- The next `for` loop iterate through the string and count how many lowercase vowels ('a', 'e', 'i', 'o', or 'u') it contains. The second for loop only finds vowels as it iterates over the original string.

- The next example demonstrates how to use a Python list as the container for a `for` loop. This example iterates over all the items in a Python list and prints each item and its corresponding Python type on a separate line.

```
>>> mylist = [1, 1.0, 1.0j, '1', (1,), [1]]
>>> for item in mylist:
...     print(item, type(item), sep="\t")
...
1       <type 'int'>
1.0     <type 'float'>
1j      <type 'complex'>
1       <type 'str'>
(1,)    <type 'tuple'>
[1]     <type 'list'>
```

V.4 Iteration and mutable containers

- The Python `list` is a *mutable* sequence, which creates an intriguing possibility: that the body of the `for` loop could modify the list it is being iterated over.

```
>>> mylist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for item in mylist:
...     if item % 2:
...         mylist.insert(0, 100)
...
^CTraceback (most recent call last):
  File "<stdin>", line 3, in ?
KeyboardInterrupt
```

- The `for` loop in this example tries to insert the number 100 at the start of the list whenever it finds an odd number in the original list. The Python interpreter will enter in an infinite loop you must interrupt by pressing Ctrl-C.

- To solve this problem, the second for loop creates a copy of the original list using the slice operator. The `for` loop then iterates through the copy, while you're modifying the original list. The end result is a modified original list that now starts with five new elements of 100.

```
>>> mylist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for item in mylist[:]:
...     if item % 2:
...         mylist.insert(0, 100)
...
>>> mylist
[100, 100, 100, 100, 100, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```


V.5 The for loop and sequence indexing

V.5.1 Standard use of the for loop

- The Python `for` loop is not limited to iterate through a collection of items. If you want to use the `for` loop to perform a series of operations a specific number of times you can generate a collection of numbers of the appropriate size using the built-in **`range()`** function.

- The **`range()`** function generates, one after the other, a series of integers. More precisely the `range()` function returns a special kind of iterator (a **`range`** object: `range` is a class of immutable iterable objects) that can be used everywhere an *iterable* is expected.

- You can give the `range()` function 1, 2 or 3 parameters.
  All parameters must be integers.
  All parameters can be positive or negative.
  When 2 or 3 parameters are given, the first parameter is the number at which range do starts and the second the number up to which the `range()` should go (**note that the second number itself is not in the range but numbers up to this number are**).
  The third parameter, if specified, is the step, i.e. the difference between successive items (if you do not specify the step the default is one).
  So, the call *`range(11,20)`* returns the integers from 11 to 19, inclusive, whereas the call *`range(12, 89, 2)`* returns a list of the even integers from 12 to 88.
  When only 1 parameter is given it correspond to the number to stop at (the first generated number is, in this case, 0, hence, the call *`range(10)`* returns one after the other the integers 0 through 9, inclusive.

```
>>> r=range(10)
>>> r
range(0, 10)
>>> for val in r:
...       print(val, end=",")
...
0,1,2,3,4,5,6,7,8,9,
>>> for val in range(2,14,2):
...       print(val, end=",")
...
2,4,6,8,10,12,
>>> for val in range(14,2,-2):
...       print(val, end=",")
...
14,12,10,8,6,4,
>>> for val in range(-2,-14,-2):
...       print(val, end=",")
...
-2,-4,-6,-8,-10,-12,
```

- You can produce a list of integers by passing the range object returned by `range()` to the `list()` built-in function.

```
>>> l=list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- The following example demonstrates how you can use the `range()` function within a `for` loop to create a times table for the integers 1 through 10, inclusive.

```
>>> for row in range(1, 11):
...       for col in range(1, 11):
...           print("{:4d} ".format(row * col), end="")
...       print()
...
   1    2    3    4    5    6    7    8    9   10
   2    4    6    8   10   12   14   16   18   20
   3    6    9   12   15   18   21   24   27   30
   4    8   12   16   20   24   28   32   36   40
   5   10   15   20   25   30   35   40   45   50
   6   12   18   24   30   36   42   48   54   60
   7   14   21   28   35   42   49   56   63   70
   8   16   24   32   40   48   56   64   72   80
   9   18   27   36   45   54   63   72   81   90
  10   20   30   40   50   60   70   80   90  100
```

- In the above example, each loop iterates over a sequence of integers; 1 to 10, inclusive.

- In the example, the string formatting specifies that a new string will be created from an integer and that three characters need to be reserved to hold the integer (if the integer is fewer than four characters, it will be left-padded with spaces, which makes things line up). The second `print` function is used to print a newline so that the next row in the times table is printed on a new line.

- The `range()` function can also be used to iterate over a container, accessing each item in the sequence by using the appropriate index.

```
>>> st = "Hello"
>>> for index in range(len(st)):
...     print(index, "-->", st[index])
...
0 --> H
1 --> e
2 --> l
3 --> l
4 --> o
```

- This example demonstrates how to use the `len()` function as the argument to the `range()` function to create a sequence of integers that can be used to access each character from the string individually.

Note: you can obtain a similar result (in a more "Pythonic" way) with the help of the built-in **`enumerate()`** function.

```
>>> st = "Hello"
>>> for index, char in enumerate(st):
...     print(index, "-->", char)
...
0 --> H
1 --> e
2 --> l
3 --> l
4 --> o
```


V.5.2 List comprehension

- A common use of a for loop is to inspect each item in a sequence and build a new list by appending the results of an expression computed on some or all of the items inspected.

- The expression form, called a ***list comprehension***, lets you code this common idiom concisely and directly.

- A list comprehension has the following syntax:

  [*expression* **for** *target* **in** *iterable lc-clauses***]**

  target and *iterable* are the same as in a regular `for` statement.

  You must enclose the *expression* in parentheses if it indicates a tuple.

  lc-clauses is a series of zero or more clauses, each with one of the following forms:

  ```
  for target in iterable
  if expression
  ```

- A list comprehension is equivalent to a for loop that builds the same list by repeated calls to the resulting list's `append()` method.

```
>>> l=[10,20,30,40]
>>> result=[x+1 for x in l]
>>> print(result)
[11, 21, 31, 41]
```

- Here's a list comprehension that uses an `if` clause:

```
>>> l=[10,20,30,40]
>>> result=[x+1 for x in l if x > 20]
>>> print(result)
[31, 41]
```

- And here's a list comprehension that uses a `for` clause:

```
>>> l1=[10,20,30,40]
>>> l2=[1,2,3,4]
>>> result=[x+y for x in l1 for y in l2]
>>> print(result)
[11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 33, 34, 41, 42, 43,
44]
```

V.6 Comparison Operators

- The comparison operators provided by Python are:

Operator	Description
`x == y`	x equals y.
`x < y`	x is less than y.
`x > y`	x is greater than y.
`x >= y`	x is greater than or equal to y.
`x <= y`	x is less than or equal to y.
`x != y`	x is not equal to y.
`x <> y`	x is not equal to y (obsolete)
`x is y`	x and y are the same object (same identity)
`x is not y`	x and y are different objects.
`x in y`	x is a member of the container (e.g., sequence) y.
`x not in y`	x is not a member of the container (e.g., sequence) y.

- All objects, including numbers, can also be compared for equality (==) and inequality (!=).

- Comparisons requiring order (**<**, **<=**, **>**, **>=**) may be used between any two numbers except complex ones.

- Comparisons can be chained in Python like this:

```
0 < age < 100
```

V.6.1 Comparing Strings and Sequences

- Strings are compared according to their order when sorted alphabetically.

- Actually, characters are sorted by their ordinal values. The ordinal value of a letter can be found with the **`ord()`** function, whose inverse is **`chr()`**.

- Other sequences are compared in the same manner: item by item starting from the left:

```
>>> [1, 2] < [2, 1]
True
```

- If the sequences contain other sequences as elements, the same rule applies to these sequence elements:

```
>>> [2, [1, 4]] < [2, [1, 5]]
True
```


V.6.2 Boolean Operators

- As described previously (see III.3), Python provides the following Boolean operators:

Operator	Description
`not x`	negation of x
`x and y`	if x is True return y else return x
`x or y`	if x is False return y else return x

VI THE DICTIONARY TYPE

- A **dictionary** data type is common to many languages. It's sometimes known as an ***associative array*** because data is associated with a key value, or as a ***hash table***.

- In Python a dictionary (`dict`) is a heterogeneous, mutable, mapping container data type.

VI.1 Creating a dictionary

- The dictionary container type in Python is different from the `tuple`, `str` and `list` container types in that it is an unordered collection.

- Instead of accessing items from the collection given an index number, you use a key value.

- To construct a dictionary container you must supply both the keys and the corresponding values.

- A dictionary is created in Python by using curly braces (**`{}`**),with key-value combinations separated by the colon character (**`:`**).

- If no key-value combinations are provided, you create an empty dictionary.

- As is the case with any container type, you can use the built-in `len()` function to find out how many items are in the collection.

```
>>> d = {0: 'zero', 1: 'one', 2: 'two', 3: 'three', 4: 'four',
5: 'five'}
>>> d
{0: 'zero', 1: 'one', 2: 'two', 3: 'three', 4: 'four', 5:
'five'}
>>> len(d)
6
>>> type(d)          # Base object is the dict class
<type 'dict'>
```

- The key isn't restricted to being an integer; it can be any non-mutable data type, including an integer, float, tuple, or string. But because a list is mutable, it can't be used as the key in a dictionary.

- The values in a dictionary, however, can be any data type.

```
>>> d = {'one': [0, 1, 2 , 3, 4, 5, 6, 7, 8, 9]}
>>> d
{'one': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}
```

- The underlying data type for a dictionary in Python is the **`dict`** object.

- The `dict` class provides several constructors to create a dictionary directly, as opposed to using the curly braces. Here are examples of the use of these constructors:

```
>>> l = [(0, 'zero'), (1, 'one'), (2, 'two'), (3, 'three')]
>>> d = dict(l)
>>> d
{0: 'zero', 1: 'one', 2: 'two', 3: 'three'}
>>> l = [[0, 'zero'], [1, 'one'], [2, 'two'], [3, 'three']]
>>> d
{0: 'zero', 1: 'one', 2: 'two', 3: 'three'}
>>> d = dict(l)
>>> d
{0: 'zero', 1: 'one', 2: 'two', 3: 'three'}
>>> d  = dict(zero=0, one=1, two=2, three=3)
>>> d
{'zero': 0, 'three': 3, 'two': 2, 'one': 1}
```

- The first and second examples demonstrate how to properly create a dictionary: in the first case by using a list where each element is tuple, and in the second case by using a list where each element is another list. In either case, the inner container is used to obtain the key-to-data value mapping.

- The third method for creating a `dict` container directly is to provide the key-to-data value mapping directly. This technique allows you to explicitly define the keys and their corresponding values.

VI.2 Accessing and modifying a dictionary

- A dictionary type is an unordered container that relies on a key-to-value mapping. As a result, items in a dictionary are accessed by a key value and not by their location within a sequence.

- To access a data value from a dictionary is nearly identical to pulling data out of any container type. You place the key value inside square brackets that follow the container name.

```
>>> d  = dict(zero=0, one=1, two=2, three=3)
>>> d
{'zero': 0, 'three': 3, 'two': 2, 'one': 1}
>>> d['zero']
0
```

- Because dictionaries are unordered, the slicing functionality that you can use with other container data types isn't available with dictionaries.

- Trying to use slicing or trying to access data from nonexistent keys throws exceptions, indicating the relevant error.

- The dictionary container in Python is also a mutable data type, which means you can change it after it has been created. You can add new key-to-data value mappings, you can change existing mappings, and you can remove mappings.

- Changing a data value is simple: assign the new value to the appropriate key.

- Adding new key-to-data value mappings is also simple: assign the relevant data to the new key value.

- You delete a mapping by using the **`del`** operator along with the key that should be deleted from the container.

```
>>> d = {0: 'zero', 1: 'one', 2: 'two', 3: 'three'}
>>> d[0]
'zero'
>>> d[0] = 'Zero'
>>> d
{0: 'Zero', 1: 'one', 2: 'two', 3: 'three'}
>>> d[4] = 'four'
>>> d[5] = 'five'
>>> d
{0: 'Zero', 1: 'one', 2: 'two', 3: 'three', 4: 'four', 5:
'five'}
>>> del d[0]
>>> d
{1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five'}
>>> d[0] = 'zero'
>>> d
{0: 'zero', 1: 'one', 2: 'two', 3: 'three', 4: 'four', 5:
'five'}
```

- It is possible to create a dictionary with heterogeneous keys and data values:

```
>>> d = {0: 'zero', 'one': 1}
>>> d
{0: 'zero', 'one': 1}
>>> d[0]
'zero'
>>> type(d[0])
<type 'str'>
>>> d['one']
1
>>> type(d['one'])
<type 'int'>
>>> d['two'] = [0, 1, 2]
>>> d
{0: 'zero', 'two': [0, 1, 2], 'one': 1}
>>> d[3] = (0, 1, 2, 3)
>>> d
{0: 'zero', 3: (0, 1, 2, 3), 'two': [0, 1, 2], 'one': 1}
>>> d[3] = 'a tuple'
>>> d
{0: 'zero', 3: 'a tuple', 'two': [0, 1, 2], 'one': 1}
```

Note: the order of the resulting dictionaries doesn't match the order in which data was inserted. Fundamentally, the order of the items in a dictionary is controlled by the actual implementation of the Python dictionary data type.

VI.3 Programming with dictionaries

- Dictionaries support the majority of operations you may be familiar with from other, simpler data types. These operations include the general relational operators such as **<**, **>**, and **==** :

```
>>> d1 = {0: 'abc'}
>>> d2 = {0: 'abd'}
>>> d1 < d2
True
>>> d2 = d1
>>> d1 < d2
False
>>> d1 == d2
True
>>> id(d1)
10239120
>>> id(d2)
10239120
>>> d2 = d1.copy()
>>> d1 == d2
True
>>> id(d1)
10238688
>>> id(d2)
10239120
```

- The previous example creates two dictionaries and uses them to test the **<** relational operator (you'll rarely compare two dictionaries in this manner; but if you need to do so, you can).

- This example then assigns the dictionary assigned to the variable d1 to the variable d2. Note that this isn't a copy operation, as demonstrated by the fact that both d1 and d2 have the same identifier values, as returned by the built-in `id()` function.

- To make a copy of a dictionary, you can use the **`copy()`** method.

- The dictionary container data type provides several built-in methods, including the **`keys()`** method as well as the **`values()`** methods. These methods return a *view* that contains the key or data values, respectively, within the calling dictionary.
  Dictionary views can be iterated over to yield their respective data, and support membership tests:

- To determine if a given key exists within a dictionary, you can simply use the **`in`** operator.

- You can do a similar operation to test whether a value is in the data values returned by calling the `values()` method.

```
>>> d = {0: 'zero', 3: 'a tuple', 'two': [0, 1, 2], 'one': 1}
>>> 0 in d
True
>>> 'zero' in d.values()
True
>>> list(d.keys())
[0, 'one', 3, 'two']
```

- A `for` loop can be used to iterate over the items in a dictionary.

- You can use the `for` loop in combination with the Python ***iterator*** returned from the **`keys()`**, **`values()`**, or **`items()`** methods.

- The `keys()` method lets you iterate through the key values of a dictionary, whereas the `values()` method allows you to iterate through the data values contained within a dictionary.

Note: *`for x in d`* is a shorthand for *`for x in d.keys()`*

```
>>> d = {0: 'zero', 3: 'a tuple', 'two': [0, 1, 2], 'one': 1}
>>> for k in d.keys():
...     print(k, "-->", d[k])
...
0 --> zero
one --> 1
3 --> a tuple
two --> [0, 1, 2]
```

- On the other hand, the `items()` method lets you iterate through the key-to-data value mappings at the same time.

```
>>> for k,v in d.items():
...     print(k, "-->", v)
...
0 --> zero
one --> 1
3 --> a tuple
two --> [0, 1, 2]
```


VI.4 Other methods

- Non-mutating methods

`items()`

Returns a view of all items (key/value pairs) in the dictionary

`get(k[,x])`

Returns dictionary[k] if *`k`* is a key in the dictionary, otherwise returns *`x`* (or `None`, if *`x`* is not given)

- Mutating methods

`clear()`

Removes all items from the dictionary

`update(dict)`

For each keys *`k`* in dict, sets *`dictionary[k]`* equal to *`dict[k]`*

`setdefault(k[,x])`

Returns *`dictionary[k]`* if *`k`* is a key in the dictionary otherwise sets *`dictionary[k]`* equal to x and returns x

`popitem()`

Removes and returns an arbitrary item (key/value pair)
The `popitem()` method can be used for destructive iteration on a dictionary.

Note: Since Python 2.5, if you derive a new dictionary class from `dict` (see IX.7), you can define a `__missing__(self, key)` function to handle retrievals when the key doesn't exist in the dictionary.

VII FUNCTIONS

VII.1 Definition

- Like most modern programming languages, Python supports the use of functions to encapsulate a set of statements that can be used over and over as necessary.

- Following presents a simple, pseudocode outline of how you can write a function in Python.

```
def myFunction(optional input data):
    initialize any local data
    actual statements that do the work
    optionally return any results
```

- The first line of a function definition, also known as the function signature, begins with **`def`** (shorthand for define).

- The function can take input parameters, which are supplied in parentheses that follow the function name. The function can also return a value (or, more formally, an object), including a Python container like a `tuple`.

- The function signature ends with a colon, which indicates that the body of the function follows on subsequent lines.

- By convention, in a function name composed of several words, the first letter of each word is capitalized except the first letter of the first word. This approach is known as ***camel casing*** and is a popular technique used in Python (and other languages) to make it easier to read the names of functions.

- The body of a function consists of a block of Python statements; they must be indented just like the body of a loop or conditional statement.

- To call a method and have it do its work -- you enter the name of the function, followed by parentheses. Between parentheses, you pass the arguments needed to initialise the input parameters, if any.

- The following function uses a `for` loop to create a times table. You can specify the number of rows and the number of columns that should be included in the generated times table:

```
>>> def timesTable(nrows=5, ncols=5):
...     for row in range(1, nrows + 1):
...         for cols in range(1, ncols + 1):
...             print(f"{row * cols:3d} ", end="")
...         print()
...
```

```
>>> timesTable(4, 6)
  1    2    3    4    5    6
  2    4    6    8   10   12
  3    6    9   12   15   18
  4    8   12   16   20   24
>>> timesTable()
  1    2    3    4    5
  2    4    6    8   10
  3    6    9   12   15
  4    8   12   16   20
  5   10   15   20   25
>>> timesTable(ncols=3)
  1    2    3
  2    4    6
  3    6    9
  4    8   12
  5   10   15
```

- In Python, a function is an object. As a result, you can assign a function to a variable and by doing so you create an alias of the function.

```
>>> t = timesTable
>>> type(t)
<type 'function'>
>>> t
<function timesTable at 0x64c30>
>>> t()
  1    2    3    4    5
  2    4    6    8   10
  3    6    9   12   15
  4    8   12   16   20
  5   10   15   20   25
```

- One other important point about the *`timesTable`* function is the presence of default values for the two input parameters.

- You provide a ***default value*** for a parameter in the function signature by appending an equal sign and the value to the parameter name, like this: *`nrows=5`*. Thanks to these defaults parameter values you can include neither, one, or both of the input parameters when you call the function.

- If you don't specify all of the parameters during a function invocation; you must explicitly name the parameters you're specifying so the Python interpreter can properly call the function. This is demonstrated in the last function invocation, which explicitly calls the `timesTable` function with *`ncols=3`*; the function creates a times table with five rows (the default value) and three columns (the supplied value).

VII.2 Returning data

- A function may return a value using the **`return`** statement.

- In Python, functions can return any valid object type, including a container type. Therefore, you can compute multiple quantities and easily return them to the calling statement.

Note: when no `return` statement is reached during execution and the end of the function is reached. The function returns **`None`**.

- The following function steps through the data (which is assumed to be a Python container holding numerical data), calculates the mean value of the set of data, then returns the value. Calling this function with no parameters, with a non-container data type, or with a container that holds non-numeric data results in an error.

```
>>> def stats(data):
...     sum = 0.0
...     for value in data:
...         sum += value
...     return (sum/len(data))
...
>>> stats([1, 2, 3, 4, 5])      # Find the mean value from a list
3.0
>>> stats((1, 2, 3, 4, 5))      # Find the mean value from a tuple
3.0
```

VII.3 Variable number of arguments

- In the function signature, at the end of the formal parameters, you may optionally use either or both of the special forms ***`identifier1`** and ****`identifier2`**.

Note: if both are present, the one with two asterisks must be last.

- ***`identifier1`** indicates that any call to the function may supply extra positional arguments. Every call to the function binds *`identifier1`* to a tuple whose items are the extra positional arguments (or the empty tuple, if there are none).

- ****`identifier2`** specifies that any call to the function may supply extra named arguments. *`identifier2`* is bound to a dictionary whose items are the names and values of the extra named arguments (or the empty dictionary, if there are none).

```
>>> def sum(*numbers):
...     result = 0
...     for number in numbers: result += number
...     return result
...
>>> sum(12,13,17)
42
```


VII.4 Variables and functions

- A function's formal parameters, plus any variables that are bound (by assignment or by other binding statements) in the function body, comprise the function's local scope.
- Each of these variables is called a ***local variable*** of the function.
- Variables that are not local are known as global variables.
- Global variables are attributes of the module object (see X.3).
- A local variable hides the global variable of the same name.
- If a function needs to rebind some global variables, the first statement of the function must be:

  `global` `identifiers`

  where *`identifiers`* is one or more global variable names separated by commas (`,`).

```
nb=0

def inc():
    global nb  # binds nb to the "global" variable nb
    nb+=1

inc()
print("Nb is", nb)
```

```
Nb is 1
```

Note: you don't need `global` if the function body simply uses (read) a global variable.

VII.5 Nested functions and nested scopes

- A **`def`** statement within a function body defines a **nested function**.
- Code in a nested function's body may access (but not rebind) local variables of an outer function.
- There is a **`nonlocal`** statement that works in the same way as the global statement, except that it is used to refer to local variables of an outer function.

VII.6 Lambda Expressions

- If a function body contains a single `return` expression statement, you may choose to replace the function with the special **`lambda`** expression form:

  `lambda` ***`parameters`*****`:`** ***`expression`***

- A lambda expression is a kind of anonymous function whose body is a single `return` statement.

- Note that the lambda syntax does not use the `return` keyword.

- You can use a lambda expression wherever you would use a reference to a function.

- Lambda can sometimes be handy when you want to use a simple function as an argument or return value.

- Here's an example that uses a `lambda` expression as an argument of the `map()` function:

```
>>> numbers = [72, 101, 108]
>>> map(lambda n: 2*n, numbers)
[144, 202, 216]
```

- In this example, you don't have to use lambda expressions, it works just fine with named functions as well.

VII.7 Generators

- When the body of a function contains one or more occurrences of the keyword **`yield`**, the function is called a ***generator***.

- When a *generator* is called, the function body does not execute. Instead, calling the generator returns a special **`iterator`** object that wraps the function body, the set of its local variables (including its parameters), and the current point of execution, which is initially the start of the function.

- When the **`next()`** built-in function is called on this `iterator` object, the function body executes up to the next `yield` statement, which takes the form:

  `yield` *`expression`*

- When a yield statement executes, the function is frozen with its execution state and local variables intact, and the expression following `yield` is returned as the result of the `next()` function.

- On the next call to `next()`, execution of the function body resumes where it left off, again up to the next `yield` statement.

- If the function body ends or executes a `return` statement, the `iterator` raises a `StopException` to indicate that the `iterator` is finished.

Note: `return` statements in a generator cannot contain expressions.

- The most common way to use an `iterator` is to loop on it with a `for` statement, you typically call a generator like this:

  `for` *`avariable`* `in` *`somegenerator`*`(`*`arguments`*`):`

- Here is a generator that works somewhat like the built-in `range()` function, but returns a sequence of floating-point values instead of a sequence of integers:

```
def frange(start, stop, step=1.0):
    while start < stop:
        yield start
        start += step

for i in frange (2.3, 6.67):
    print(i)
```

```
2.3
3.3
4.3
5.3
6.3
```

- In languages that provide for generators, an important feature is the ability to pass a value back into the generator. This allows for supporting a programming feature called ***coroutines***.

- In order to make the generators more powerful, the designers of Python have added in Python 2.5 the ability to pass data back into the generator.

- If, from the perspective of the generator, you think of the `yield` as calling something, then the concept is easy: You just save the results of `yield`:

  ```
  x = yield start
  ```

- From the perspective of the caller of the generator, this statement returns control back to the caller, just as before. From the perspective of the generator, when the execution comes back into the generator, a value will come with it (in this case, the generator saves it into the variable x).

- Where does the value come from? The caller calls a function named `send()` to pass a value back into the generator. The function `send()` behaves just like the function `next()`, except that it passes a value.

- **`iterator`** objects also have a `close()` method. This method frees up the resources for the generator.
  If you call `next()` again after calling `close()`, you'll get a `StopIteration` exception.

VII.8 Built-in Functions

- This section describes some interesting Python functions available in module **`__builtin__`** that have not been covered so far.

`abs(x)`

Returns the absolute value of number x.

`callable(obj)`

Returns True if *`obj`* can be called, otherwise False.

`coerce(x,y)`

Returns a pair whose two items are numbers *`x`* and *`y`* converted to a common type.

`compile(string, filename, kind)`

Compiles a string and returns a code object usable by exec or eval.
When string is a multiline compound statement, the last character must be '\n'. *`kind`* must be 'eval' when string is an expression and the result is meant for eval, otherwise kind must be 'exec'.
`filename` must be a string, and is used only in error messages (if and when errors occur).

`dir([obj])`

Called without arguments, `dir()` returns the sorted list of all variable names that are bound in the current scope.
`dir(obj)` returns the sorted list of all names of attributes (including ones that are inherited) of *`obj`* and from its type.

`eval(expr,[globals[,locals]])`

Returns the result of an expression. *`expr`* may be a code object ready for evaluation or a `string`. In the case of a string, `eval` gets a code object by calling `compile(expr, 'string', 'eval')`. eval evaluates the code object as an expression, using the `globals` and `locals` dictionaries as namespaces. When both arguments are missing, `eval` uses the current namespace. `eval` cannot execute statements; it can only evaluate expressions.

`filter(func,seq)`

Constructs a list from those elements of *`seq`* for which *`func`* is true. *`func`* can be any callable object that accepts a single argument or `None`. *`seq`* must be a sequence, or an *`iterable`* object.
`filter()` calls *`func`* on each item of *`seq`* and returns the list of items for which func's result is true.

hex(x)

Converts integer *x* to a hexadecimal string representation.

iter(obj)
iter(func,sentinel)

Creates and returns an `iterator`: an object with a `next()` method that you can call repeatedly to get one item at a time. When called with one argument, `iter(obj)` normally returns `obj.__iter__( )`.

When called with two arguments, the first argument must be callable without arguments, and `iter(func,sentinel)` is equivalent to the following simple generator:

```
def iterSentinel(func, sentinel):
    while 1:
        item = func(  )
        if item == sentinel: raise StopIteration
        yield item
```

The statement `for x in obj` is equivalent to `for x in iter(obj)`.

locals()

Returns a read-only dictionary that represents the current local namespace.

map(func,seq,*seqs)

Applies *func* to every item of *seq* and returns a list of the results.
`map(func, seq)` is the same as: `[func(item) for item in seq]`
When *func* is `None`, map returns a list of tuples, each with n items (one item from each iterable).

max(s,*args)
min(s,*args)

Returns the largest (the smallest) item in the only argument *s* (*s* must be iterable) or the largest (smallest) of multiple arguments.

next(iterator[, default])

Retrieve the next item from the iterator by calling its `__next__()` method. If *default* is given, it is returned if the iterator is exhausted, otherwise `StopIteration` is raised.

oct(x)

Converts integer *x* to an octal string representation.

pow(x,y[,z])

When z is present, `pow(x,y,z)` returns `x**y%z`.
When z is missing, `pow(x,y)` returns `x**y`.

round(x,n=0)

Returns a float whose value is number `x` rounded to `n` digits after the decimal point.

sorted(iterable[, key][, reverse])

Return a new sorted list from the items in *`iterable`*.

`key` specifies a function of one argument that is used to extract a comparison key from each list element: *`key=str.lower`*. The default value is None (compare the elements directly).

`reverse` is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

vars([obj])

When called with no argument, `vars()` returns a read-only dictionary that represents all variables that are bound in the current scope (like `locals()`, covered in this section). `vars(obj)` returns a dictionary that represents all attributes currently bound in *`obj`*, as covered in dir in this section.

zip(seq,*seqs)

Returns a list of tuples, where the n^{th} tuple contains the n^{th} element from each of the argument sequences. `zip()` is called with n `iterable` objects as arguments. If the `iterable` objects have different lengths, `zip()` returns a list as long as the shortest iterable, ignoring trailing items in the other iterable objects.

```
>>> names = ['anne', 'beth', 'george', 'damon']
>>> ages = [1, 45, 32, 102]
>>> var = list(zip(names, ages))
>>> var
[('anne', 12), ('beth', 45), ('george', 32), ('damon', 102)]
>>> for name, age in var:
...         print(f'{name} is {age} {"years" if age>1 else\
"year"} old')

anne is 1 year old
beth is 45 years old
george is 32 years old
damon is 102 years old
```

VIII EXCEPTIONS

VIII.1 What is an exception

- Like many other programming languages, Python has exception handling via **`try...except`** blocks.

- Exceptions are everywhere in Python. Virtually every module in the standard Python library uses them, and Python itself will raise them in a lot of different circumstances.

- For instance:

  Accessing a non-existent dictionary key will raise a `KeyError` exception.
  Searching a list for a non-existent value will raise a `ValueError` exception.
  Calling a non-existent method will raise an `AttributeError` exception.
  Referencing a non-existent variable will raise a `NameError` exception.
  Mixing datatypes without coercion will raise a `TypeError` exception.
  …

- When using in the Python IDE, if an exception is raised, the exception is printed (generally in red), and that is it. This is called an ***unhandled*** exception.

- In the following example, the file constructor, is used here to try to open a file for reading. But the file doesn't exist, so this raises the **`IOError`** exception.

- Since you haven't provided any explicit check for an `IOError` exception, Python just prints out some debugging information about what happened and then gives up.

```
>>> input = file("/non-existing-file", "r")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: '/non-existing-
file'
```

- If an exception is raised inside a function, and isn't handled there, it propagates to the place where the function was called. If it isn't handled there either, it continues propagating until it reaches the main program (the global scope), and if there is no exception handler there, the program halts with an error message and some information about what went wrong (a stack trace).

- Exceptions, when raised, can be ***handled***. Sometimes an exception is really because you have a bug in your code (like accessing a variable that doesn't exist), but many times, an exception is something you can anticipate. If you're opening a file, it might not exist. If you're connecting to a database, it might be unavailable, or you might not have the correct security credentials to access it, …

VIII.2 Handling exceptions

- If you know a line of code may raise an exception, you should handle the exception using a `try...except` block.

- In the following example, when the `file` constructor raises an `IOError` exception, the `except IOError:` line catches the exception and executes your own block of code, which in this case just prints a more pleasant error message.

- Once an exception has been handled, processing continues normally on the first line after the `try...except` block.

```
>>> try:
...         input = open("/non-existing-file", "r")
... except IOError:
...         print("The file does not exist, exiting gracefully")
...         import sys
...         sys.exit(1)
The file does not exist, exiting gracefully
```

- In the previous example, because the `except` clause only looked for `IOError` exceptions, any other exception that occurs in the `try` block will not be handled and will halt the program. To catch several kinds of exceptions, you can simply add other `except` clauses to the same `try…except` statement:

```
>>> try:
...         input = open("/non-existing-file", "r")
... except IOError:
...         print("The file does not exist")
... except Exception:
...         print("Another exception …")
```

- If you want to catch more than one exception type with one block, you can specify them all in a `tuple`, as follows:

```
...
... except (ZeroDivisionError, TypeError):
...
```

Note: If you want to catch all exceptions in a piece of code, you can simply omit the exception class from the `except` clause.

```
>>> try:
...         input = open("/non-existing-file", "r")
... except:
...         print("Problem ...")
...
```


- If you want access to the exception itself in an except clause, you can use the **as** keyword: *exception-type* ***as*** *exception-variable*

```
>>> try:
...          input = open("/non-existing-file", "r")
... except IOError as e:
...          print("The file does not exist")
...          print(e)

The file does not exist
[Errno 2] No such file or directory: '/non-existing-file'
```

- A `try...except` block can have an **`else`** clause, like an `if` statement. If no exception is raised during the `try` block, the `else` clause is executed afterwards.

- There are other uses for exceptions besides handling actual error conditions. A common use in the standard Python library is to try to import a module, and then check whether it worked.

- Importing a module that does not exist will raise an `ImportError` exception. You can use this to define multiple levels of functionality based on which modules are available at run-time, or to support multiple platforms (where platform-specific code is separated into different modules).

- The following example print different messages depending on which OS you use. To do so the presence of an OS dependent module (`termios` for Unix, `msvcrt` for windows) is being checked.

```
>>> try:
...       import termios
... except ImportError:
...       try:
...                import msvcrt
...       except ImportError:
...                print("Unknown OS")
...       else:
...                print("Windows")
... else:
...       print("Unix/Linux")
```


VIII.3 try..finally and try..except..finally

- Finally, there is the `finally` clause.

- You can use `try…finally` if you need to make sure that some code (for example, cleanup code) is executed regardless of whether an exception is raised or not. This code is then put in the `finally` clause.

```
try:
   # some code here
finally:
   # the code here will always be executed
```

- Since Python 2.5, you can have both `except` clauses and a `finally` clause in the same `try` statement:

```
try:
   # some code here
except SomeException:
   # the code here is executed if a SomeException is raised
except OtherException:
   # the code here is executed if a OtherException is raised
...
else:
   # the code here is executed when no exception is raised
finally:
   # the code here will always be executed
```

VIII.4 The raise Statement

- To raise an exception, you use the **`raise`** statement with an argument that is either the `Exception` class (or one of its sub-classes) or an instance of one of these classes.

```
>>> def factorial(nb):
...     res=1
...     if (nb <0):
...         raise Exception("Bad argument")
...     # Here we compute the factorial into res …
...     return res
...
>>> try:
...     result=factorial(5)
...     print(result)
...     result=factorial(-6)    # exception raised here
...     print(result)                   # never reached
... except Exception:
...     print("Problem ...")
```

- There are actually two other ways to use `raise`:
  1. The argument may be a string, but this is considered obsolete and you get a deprecation warning if you use this possibility.
  2. If you have caught an exception but you want to raise it again, you can call `raise` without any arguments.

- There are many built-in exception classes available, they are all found in the module **`exceptions`** (to list the content of a module, you can use the **`dir()`** function).

- Here are a few examples of built-in exception classes:

`BaseException`	The root class for all exceptions
`Exception`	A direct subclass of `BaseException`. The root class for most exceptions
`KeyboardInterrupt`	A direct subclass of `BaseException`. Raised when Ctrl-c is being pressed.
`SystemExit`	A direct subclass of `BaseException`. Raised to force the program to exit.
`AttributeError`	Raised when attribute reference or assignment fails
`IOError`	Raised when trying to open a nonexistent file (among other things)
`IndexError`	Raised when using a non-existent index on a sequence
`KeyError`	Raised when using a non-existent key on a mapping
`TypeError`	Raised when a built-in operation or function is applied to an object of the wrong type
`ZeroDivisionError`	Raised when the second argument of a division or modulo operation is zero

- You can also define your own exceptions by creating a class that inherits from the built-in `Exception` class (either directly or indirectly, which means that sub-classing any other built-in exception is okay), and then raise your exceptions with the `raise` command.

- Thus, writing a custom exception basically amounts to something like this:
  `class SomeCustomException(Exception): pass`

IX OBJECT-ORIENTED PROGRAMMING

IX.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.

IX.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.

IX.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`** .

IX.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).

IX.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.

IX.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)
```

IX.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
```

IX.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.

IX.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.

IX.4 Class-Level Methods

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

IX.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")
```


IX.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")
```


IX.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))
```


IX.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`*).

IX.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.

IX.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__`.

IX.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.

IX.5.4 Attribute Reference Basics

- An attribute reference is an expression of the form `x.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 `x.name` to refer to an attribute of instance `x`, the lookup proceeds in two steps:

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

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

IX.5.5 The __new__() method

- Each new-style 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.

IX.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 syntax. 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`*).

IX.7 Inheritance

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

- The new-style object model, like the classic one, 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 new-style class only subclasses at most one substantial built-in type.

IX.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 new-style 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

IX.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.

IX.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
```

IX.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 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

IX.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 IX.11.1.2.

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

IX. 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
```

IX.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.

IX.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.

IX.11.1 General-Purpose Special Methods

IX.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.

IX.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.

IX.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)`.

IX.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.

IX.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)`.

IX.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.

IX.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...])`*.

IX.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.

IX.11.2.1 Container special methods

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

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

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

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

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

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

`__iter__(self)`

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

`__len__(self)`

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

`__setitem__(self,key,value)`

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

IX.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

IX.12 Metaclasses

IX.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.

IX.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`.

IX.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.

IX.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.

IX.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`**

X 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.

X.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 X.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.

X.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.

X.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 prefix.

- 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).

X.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.

X.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.

X.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`**.

X.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`.

X.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__`**.

X.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.

X.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.

X.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`**).

X.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.

XI SIMPLE INPUT/OUTPUT

XI.1 Overview

- Throughout this manual, you've written (output) data using the **`print()`** function, which, by default, writes the expression as a string to the screen (or console window).

- The following examples demonstrate several possible use of the `print` function.

- This example outputs a simple string.

```
>>> print("Hello World!")
Hello World!
>>> print("The total value is = $", 40.0*45.50)
The total value is = $ 1820.0
```

- This example creates and outputs a compound string, created using the string formatting technique.

```
>>> print(f"The total value = ${40.0*45.50}")
The total value = $1820.00
```

- The next example creates a ***`file-like`*** object (using the `open()` built-in function), passing in the name "*`testit.txt`*" and a '*`w`*' character (to let you write to the file).

- You then use the `print()` function with the keyword argument **`file`** to write the same strings. This time, however, the data isn't displayed on the screen but stored within the file *`testit.txt`*.

- Then this file is closed. This is important in Python programs because file input and output are, by default, buffered; data isn't written as soon as you call a print function but is instead written in chunks. The simplest mechanisms for telling Python to write your data to the file is to explicitly call the **`close()`** method.

```
>>> myfile = open("testit.txt", 'w')
>>> print("Hello World!", file=myfile)
>>> print(f"The total value = ${40.0*45.50}", file=myfile)
>>> myfile.close()
```

XI.2 File-like objects (streams)

- The built-in function **`open(file,mode='r', ...)`** open the *`file`* and returns a corresponding file-like object (an object exposing a file-oriented API). This is the basic mechanism by which you interact with files on your computer.

- You can use a file-like object to read data, to write data, to append data to a file, and to work with either binary or textual data.

- The `open()` function allows you to specify an opening mode:

  - The mode can be '**`r`**', '**`w`**' or '**`a`**' for reading (the default), writing or appending.

  - The file will be created if it doesn't exist when opened for writing or appending; it will be truncated when opened for writing.

  - The mode can also be '**`x`**': '`x`' is similar to '`w`'. But with '`x`', if the file exists, it raises `FileExistsError`.

  - On Windows, you must add a '**`b`**' to the mode for binary files.

  - You can add a '**`+`**' to the mode to allow simultaneous reading and writing.

- A third optional argument can be provided: the **`buffering`** argument. If it is 0 it means the file is unbuffered, 1 means line buffered, and larger numbers specify the buffer size.

XI.3 Reading data from a file

- To read data from a file, you first create an appropriate file object and read the contents using the **`read()`** method. This method reads the entire file into a string.
  If you try to read the contents again, you're at the end of the file, so nothing can be read.
  The solution to this problem is to go back to the beginning of the file using the **`seek()`** method, which takes a single parameter that indicates where in the file you want to start reading or writing (for example, zero indicates the start of the file).

```
>>> myfile = open("testit.txt")
>>> content = myfile.read()
>>> print(content)
Hello World!
The total value = $1820.00
>>> myfile.seek(0)
>>> content = myfile.read()
>>> print(content)
Hello World!
The total value = $1820.00
>>> content.split()
['Hello', 'World!', 'The', 'total', 'value', '=', '$1820.00']
>>> content.split('\n')
['Hello World!', 'The total value = $1820.00', '']
>>> for line in content.split('\n'):
...     print(line)
...
Hello World!
The total value = $1820.00

>>> myfile.close()
```

- It is also possible to read a file line by line, thanks to the **`readlines()`** method. This method reads the entire file into memory and splits the contents into a list of lines, you can then use a `for` loop to iterate over the list of strings, printing them out one at a time:

```
>>> myfile = open("testit.txt")
>>> for line in myfile.readlines():
...     print(line)
...
Hello World!

The total value = $1820.00

>>> myfile.close()
```

- It is also possible to use an implicit variable for the file object combined with the `readlines()` method:

```
>>> for line in open("testit.txt").readlines():
...     print(line)
...
Hello World!

The total value = $1820.00
```

- The last versions of Python simplifies things even more and give the ability to iterate directly over a file-like object. In this case, you create an implicit file object, and Python does the rest, allowing you to iterate over all the lines in the file:

```
>>> for line in open("testit.txt"):
...     print(line)
...
Hello World!

The total value = $1820.00
```

- If you prefer to read the file one line after the other you can use the **`readline()`** method. When you reach the end of the file, the `readline()` method returns an empty string.

```
>>> myfile = open("testit.txt")
>>> myfile.readline()
'Hello World!\n'
>>> myfile.tell()
13L
>>> myfile.readline()
'The total value = $1820.00\n'
>>> myfile.readline()
''
>>> myfile.seek(0)
>>> myfile.readlines(100)
['Hello World!\n', 'The total value = $1820.00\n']
>>> myfile.tell()
40L
>>> myfile.readline()
''
>>> myfile.tell()
40L
>>> myfile.seek(0)
>>> myfile.read(17)
'Hello World!\nThe '
>>> myfile.seek(0)
>>> myfile.readlines(100)
['Hello World!\n', 'The total value = $1820.00\n']
>>> myfile.close()
```


- The **`tell()`** method can be used to display where you are in the file.

- You can pass a parameter to the `read()` or `readline()` methods to control how many characters are read. This parameter corresponds to the number of characters that will be read from the file.

XI.4 Writing data

- To write data to a file, you have to create a file object using one of the writing mode ('`w`' , '`r+`', …).

- Then you can use the **`write()`** method to write data into the file.

```
>>> mydata = ['Hello World!', 'The total value = $1820.00']
>>> myfile = open('testit.txt', 'w')
>>> for line in mydata:
...     myfile.write(line + '\n')
...
>>> myfile.close()
>>> myfile = open("testit.txt")
>>> myfile.read()
'Hello World!\n
The total value = $1820.00\n'
>>> myfile.close()
```

- You can also write data into a file with the help of the **`print()`** function.

```
>>> mydata = ['Hello World!', 'The total value = $1820.00']
>>> myfile = open('testit.txt', 'w')
>>> for line in mydata:
...     print(line, file=myfile)
...
>>> myfile.close()
```


XI.5 Working with binary data

- They are situations where you want to store within files binary data: integers, compressed text, float, …. You can easily do so in Python by appending 'b' to the file mode when you create the file-like object.

- When you open a file in binary mode ('bw', 'br', 'br+', ...), then you are essentially working with the bytes type.

- When you write to the file, you need to pass a `bytes` object, and when you read from it, you get a `bytes` object. In contrast, when opening the file in text mode, you are working with `str` objects. So, writing "binary" data is really writing a bytes string:

```
f=open(fileName, 'br+')
f.write(b'\x07\x08\x07')
```

- If you have actual integers you want to write as binary, you can use the `bytes()` function to convert a sequence of integers into a bytes object:

```
lst = [7, 8, 7]
f.write(bytes(lst)) # <=> f.write(b'\x07\x08\x07')
```

- But `bytes()` will only accept a sequence of numbers that actually fit in a byte, i.e. numbers between 0 and 255!. Instead, you can use the **`struct`** module to convert Python data into binary strings, which can then be written to the file.

- In C and Fortran, for instance, an integer is a raw 32-bit value
  Python usually don't use raw values; there is no guarantee that things like lists are stored contiguously in memory. So programs need to pack data into contiguous bytes for writing and unpack those bytes to recreate the structures when reading.

- Packing looks a lot like formatting a string, a format specifies the data types being packed (including sizes, where appropriate). This format exactly determines how much memory is required by the packed representation.

- The result of packing is a chunk of bytes stored as a Python string, but it is not a string of characters.

- Unpacking reverses this process; it reads bytes from a "string" according to a format, and uses the data in these bytes to create Python data structures returned as a tuple of values.

- You can use the Python's **`struct`** module to pack and unpack data.

  `pack(fmt, v1, v2, …)` packs the values *`v1`*, *`v2`*, etc. according to the format *`fmt`*, returning a `bytes`.
  `unpack(fmt, str)` unpacks the values in *`bytes`* according to *`fmt`*, returning a `tuple`.
  `calcsize(fmt)` calculates how large (in bytes) the data produced using *`fmt`* will be.

- The format string is composed of the following format specifiers:

Format specifiers	Format Meaning
c	Single character (i.e., string of length 1)
B	Unsigned 8-bit integer
h	Short (16-bit) integer
i	32-bit integer
f	32-bit float
d	Double-precision (64-bit) float
2	String of fixed size (see below)

- Any format can be preceded by a count ("4i" for instance is four integers).

- In this example, a file object is being created, then two integers are stored within it after having being packed. After having written all the data, the file is being closed and reopened for reading, again using the binary mode flag. The two integers are unpacked and displayed.

```
>>> import struct
>>> myfile = open("testit.txt", "wb")
>>> x=10
>>> y=20
>>> inputBytes=struct.pack( "ii", x, y)
>>> myfile.write(inputBytes)
>>> myfile.close()
>>> myfile = open("testit.txt")
>>> temp=myfile.read()
>>> output=struct.unpack("ii",temp)
>>> print(output)
(10,20)
>>> myfile.close()
```


XI.6 The with statement

- Python 2.5 introduces a new statement called **`with`**.

- In Python, the purpose of `with` is to let you set up a block of code that runs under a certain ***context***. When that context finishes after your code runs, it runs some cleanup code.

- A common context example is that of a file being open. You can open a file, write to the file, and close the file. While writing to the file, you can think of your code as operating under the context of the file. The context, then, would open the file and, when you finish working with it, can handle the task of closing the file for you.

- To create a context, a class must include a `__enter__()` and `__exit__()` methods.

- The `__enter__()` method, contains the code that initializes the context (such as opening a file, or starting a database transaction), and the `__exit__()` method either handle any exceptions or just perform the cleanup code.

- Several Python modules and classes have context management built in. To use them, you must use the new `with` keyword.

- To read a file, you can then, for instance, write something like:

```
with open('/data.txt', 'r') as f:
   for line in f:
       process(line)
```

- By using the `with` statement, you don't need to worry about closing the file. The code does it for you.

XI.7 The future statement

- A ***future statement*** is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python:

  `from __future__ import` ***`feature`***

- It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

- A future statement must appear near the top of the module. The only lines that can appear before a future statement are the docstring and comments.

- The only feature recognized by Python 3.6 is "generator_stop" (a feature that will be integrated in Python 3.7).

XII WHAT'S NEW IN PYTHON 2.6 AND PYTHON 3.0

- In 2009, the Python Software Foundation has released two versions of Python in the same timeframe: version 2.6 and version 3.x.

- These two are friend releases: Python 2.6 is not only the next advancement in the Python 2.x series, it is also a transitional release, helping developers begin to prepare their code for Python 3.0.

- As such, many features are being backported from Python 3.0 to 2.6.
  Python 2.6 includes additions from 3.0 that don't conflict with 2.x rules and syntax.

- 2.6 feature a warnings mode, so it can tell you how to write your python 2.6 code so it can be translated non-ambiguously to python 3.0.

- Thus, it makes sense to release both versions in at the same time.

- This chapter describes the new features introduced in Python 2.6 and Python 3.0, comparing to what existed in Python 2.5.

XII.1 Python 3.0

XII.1.1 New print() function

- The `print` statement has been replaced with a **`print()`** function, with keyword arguments to replace most of the special syntax of the old `print` statement.

```
print "Val is", val
3.0 => print ("Val is", val)
```

```
print x,
3.0 => print(x, end=" ")
```

```
print >>sys.stderr, "fatal error"
3.0 => print("fatal error", file=sys.stderr)
```

- You can also customize the separator between items:

```
print(x, y, sep=",")
```

Note: the `2to3` source-to-source conversion tool, converts automatically all print statements to print() function calls.

XII.1.2 No more old style classes

- In 3.0 old style classes do not exists anymore.

- Whether you make it inherit from the class object or not the classes you are going to create will be considered as new style classes.

XII.1.3 Default comparisons changed

- The operator <, <=, >, >= are not provided by default anymore. They raise a TypeError if you use them.

Note: == and != are still available and compare object's identities.

Note: <> is not supported anymore

Note: != now returns the opposite of ==, unless == returns `NotImplemented`.

XII.1.4 Int/Long unification

- There is only one built-in integral type, named **`int`**; but it behaves like the old `long` type, with the exception that the literal suffix `L` is neither supported by the parser nor produced by `repr()` anymore.

- `sys.maxint` is removed since `the` int type has no maximum value anymore.

XII.1.5 Sorting facilities

- `builtin.sorted()` and `list.sort()` no longer accept the *cmp* argument providing a comparison function. Use the *key* argument instead.

XII.1.6 Int divisions

- Int division always return a float (`1/2` returns a float)
- You can use **`//`** (`1//2`) to get the truncating behaviour.

Note: you can obtain the same effect in 2.x using the directive:

```
from __future__ import division
```

XII.1.7 Changes in dictionaries

- The `dict` methods `dict.keys()`, `dict.items()` and `dict.values()` return dict views (a lightweight object that can be iterated repeatedly) instead of lists.
- The `dict.iterkeys()`, `dict.itervalues()` and `dict.iteritems()` methods have been removed.
- `dict.has_key()` is removed, use the `in` operator instead.

XII.1.8 Strings and Bytes

- There is only one string type; its name is `str` but its behaviour and implementation are like `unicode` in 2.x.
- The `basestring` superclass has been removed.
- There is a new type, **`bytes`**, to represent binary data.
  A `bytes` correspond to a mutable sequence of small ints (0-255).
- Bytes literals, (b"abc", b"\xA2\xA2", b"\034\075\023", …) , create bytes instances.
- The `str` and `bytes` types cannot be mixed; you must always explicitly convert between them, using the `str.encode()` (str -> bytes) or `bytes.decode()` (bytes -> str) methods.
- All backslashes in raw strings are interpreted literally. This means that Unicode escapes ('\U' and '\u') are not treated specially.
- UTF-8 is now the default source encoding (in 2.5 the default encoding is ASCII).
- The syntax of identifiers in Python is based on the Unicode standard (in 2.5 only ASCII identifiers are valid). However, the standard library remains ASCII-only.

XII.1.9 New String formatting facilities

- A new system for built-in string formatting operations replaces the `%` string formatting operator.

- The built-in `str` class (and also the `unicode` class in 2.6) gain a new method, **`format()`**, which takes an arbitrary number of positional and keyword arguments:

```
"The story of {0}, {1}, and {c}".format(a, b, c=d)
```

- Within a format string, each positional argument is identified with a number, starting from zero, so in the above example, 'a' is argument 0 and 'b' is argument 1. Each keyword argument is identified by its keyword name, so in the above example, 'c' is used to refer to the third argument.

- Each field can also specify an optional set of 'format specifiers' which can be used to adjust the format of that field.
- Format specifiers follow the field name, with a colon (':') character separating the two:

```
"My name is {0:8}".format('Fred')
```

- There is also a global built-in function, **`format()`** which formats a single value:

```
print(format(10.0, "7.3g"))
```

XII.1.10 New I/O library

- New I/O Implementation. The API is nearly 100% backwards compatible, but completely re-implemented (currently mostly in Python). Also, binary files use `bytes` instead of strings.

- The new library supports facilities such as: Buffering, Unicode encoding/decoding, CRLF/LF mapping, ..

- The `StringIO` and `cStringIO` modules are gone. Instead, import **`io.StringIO`** or **`io.BytesIO`**.

XII.1.11 Exceptions

- Exceptions must derive from **`BaseException`**. This is the root of the exception hierarchy.

- `StandardError` is removed (already in 2.6).

- Sequence behaviour (indexing and slicing to access the arguments passed to the constructor) and the `message` property of exception instances have been removed.

- You must now use **`raise Exception(args)`** instead of `raise Exception, args`. To raise an exception.

- You must now use **`except`** `SomeException as identifier`: instead of `except Exception, identifier` when you catch an exception.

XII.1.12 Metaclasses

- The current method for specifying a metaclass is by assignment to the special variable `__metaclass__`

- In 3.0, the syntax for specifying a metaclass is via a keyword argument in the list of base classes:

```
class Foo(base1, base2, metaclass=mymeta):
    ...
```

- Additional keywords will also be allowed here, and will be passed to the metaclass.

XII.1.13 Function annotations

- Python 3.0 offers a standardized way of annotating a function's parameters and return values.

- Function annotations, both for parameters and return values, are completely optional.

- Function annotations are nothing more than a way of associating arbitrary Python expressions with various parts of a function at compile-time. By itself, Python does not attach any particular meaning or significance to annotations. The only way that annotations take on meaning is when they are interpreted by third-party libraries. These annotation consumers can do anything they want with a function's annotations. For example, one library might use string-based annotations to provide improved help messages, another library might be used to provide typechecking for Python functions and methods (this library could use annotations to indicate the function's expected input and return types), …

- Annotations for parameters take the form of optional expressions that follow the parameter name:

```
def foo(a: expression, b: expression = 5):
    ...
```

- Annotations always precede a parameter's default value and both annotations and default values are optional. All annotation expressions are evaluated when the function definition is executed, just like default values.

- Annotations for return value take the following form:

```
def sum() -> expression:
    ...
```

Note: lambda's syntax does not support annotations.

- Once compiled, a function's annotations are available via the function's **`func_annotations`** attribute. This attribute is a mutable dictionary, mapping parameter names to an object representing the evaluated annotation expression.

- There is a special key in the `func_annotations` mapping, "`return`". This key is present only if an annotation was supplied for the function's return value.

- The `pydoc` module will be updated to display the function annotations when displaying help for a function. The `inspect` module will be updated to support annotations.

XII.1.14 Class decorators

- Class decorators is an extension to the function and method decorators

- The semantics and design goals of class decorators are the same as for function decorators; the only difference is that you're decorating a class instead of a function. The following two snippets are semantically identical:

```
class A:
        pass
A = foo(bar(A))
```

```
@foo
@bar
class A:
  pass
```

XII.1.15 New, Improved, and Deprecated Modules

- The `cPickle` module is replaced by the module `pickle`.

- The `imageop`, `audiodev`, `Bastion`, `bsddb185`, `exceptions`, `linuxaudiodev`, `md5`, `MimeWriter`, `mimify`, `new`, `popen2`, `rexec`, `sets`, `sha`, `stringold`, `strop`, `sunaudiodev`, `timing`, and `xmllib` modules are gone.

- The functions **`os.tmpnam()`**, **`os.tempnam()`** and **`os.tmpfile()`** have been removed in favor of the `tempfile` module.

- The `tokenize` module has been changed to work with bytes.

XII.1.16 Other modifications

- Backticks have been remove (use repr() instead).

- `__getslice__()` and friends disappear in favour of `__getitem__()` and friends. `a[i:j]` now translates to `a.__getitem__(slice(i, j))` (or `__setitem__()` or `__delitem__()`, depending on context).

- Named parameters occurring after *args in the parameter list must be specified using keyword syntax in the call. You can also use a bare * in the parameter list to indicate that you don't accept a variable-length argument list, but you do have keyword-only arguments.

- Using **`nonlocal`** `x` you can now assign directly to a variable in an outer (but non-global) scope.

- **`raw_input()`** is renamed to `input()`. To get the old behavior of `input()`, use `eval(input())`.

- **`xrange()`** is renamed to `range()`, so `range()` will no longer produce a list but an iterable yielding integers when iterated over.

- The method **`next()`** renamed to `__next__()`, a new builtin `next()` is provided to call the `__next__()` method on an object.

- Python 3.0 introduces new octal literals; binary literals and bin().
  Instead of 0666, you write 0o666. The `oct()` function is modified accordingly. Also, 0b1010 equals 10, and bin(10) returns "0b1010".
  0666 is now a SyntaxError.

- Python 3.0 extends Iterable Unpacking. You can now write things like `a, b, *rest = some_sequence`.
  And even `*rest, a = stuff`. The `rest` object is always a list; the right-hand side may be any iterable.

- You can now invoke **`super()`** without arguments and the right class and instance will automatically be chosen. With arguments, its behavior is unchanged.

- `zip()`, `map()` and `filter()` now return iterators.

- `string.letters` and its friends (`string.lowercase` and `string.uppercase`) are gone. Use **`string.ascii_letters`** etc. instead.

- The `__oct__()` and `__hex__()` special methods are removed – `oct()` and `hex()` use **`__index__()`** now to convert the argument to an integer.

- The special method `__nonzero__()` is now renamed **`__bool__()`**.

XII.2 Python 2.6 new features

- Raising a string exception now triggers a `TypeError`.
- Attempting to catch a string exception raises `DeprecationWarning`.
- The property `BaseException.message` has been deprecated.
- A new built-in type appears: **`bytes`**.
- It is now possible to use relative imports from the main module.
- New way to represent integer literal (see XII.1.17).
- New package: **`multiprocessing`**. The processing package mimics the standard library threading module functionality to provide a process-based approach to threaded programming allowing end-users to dispatch multiple tasks that effectively side-step the global interpreter lock.
- New modules in the standard library: **`json`**
- Deprecated modules and functions in the standard library:

  - buildtools
  - cfmfile
  - commands.getstatus()
  - macostools.touched()
  - md5
  - MimeWriter
  - mimify
  - popen2, os.popen()
  - posixfile
  - sets
  - sha

- Modules removed from the standard library:

  - gopherlib
  - rgbimg
  - macfs

- Warnings for features removed in Py3k:

  - built-ins, backticks and <>, …

- Other major features:

  `with`/**`as`** are keywords
  a `__dir__()` special method to control `dir()` is added

- New set literals:

  {1, 2, 3} is the same as set([1, 2, 3])
  No empty set literal; use set()
  No frozenset literal; use frozenset({…})

- Removed built-ins:

  `compile()` and `intern()` (put in sys),
  `coerce()` (no longer needed),
  `execfile()`, `reload()` (use `exec()` instead)
  `reduce()` (put in `functools`),
  `xrange()` (use `range()` instead)

XIII INDEX

{

|

~

+

<

=

>

0

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z