Python for Image Processing

BIMU3930

Görüntü İşleme

gitlab.com/deshalb/bimu3930

Deniz Alp Durmaz

denizalpd@ogr.iu.edu.tr

Contents

Contents

Contents

- Programming Languages
- Language Processors
- Programming Paradigms

Contents

Contents

Contents

- Programming Languages
- Language Processors
- Programming Paradigms
- The Static/Dynamic Distinction
- Static Type Checking
- Dynamic Type Checking

Contents

Contents

- Python

Contents

- Python
    - 2 or 3

Contents

- Python
    - 2 or 3
    - Download Python

Contents

Contents

- Python
  - 2 or 3
  - Download Python
  - Interpreter
  - Installing Python Modules

Contents

- Python
    - 2 or 3
    - Download Python
    - Interpreter
    - Installing Python Modules
    - Using Python as a Calculator

Contents

- Python
    - 2 or 3
    - Download Python
    - Interpreter
    - Installing Python Modules
    - Using Python as a Calculator
    - Control Flow Tools

Contents

Contents

Contents

- Python
  - 2 or 3
  - Download Python
  - Interpreter
  - Installing Python Modules
  - Using Python as a Calculator
  - Control Flow Tools
  - Defining Functions
  - Coding Style
  - Data Structures

Contents

Programming Languages

Programming languages are notations for describing computations to people and to machines.

Programming Languages

Programming languages are notations for describing computations to people and to machines.

All the software running on all the computers was written in some programming language.

Programming Languages

Programming languages are notations for describing computations to people and to machines.

All the software running on all the computers was written in some programming language.

Before a program can be run, it first must be translated into a form in which it can be executed by a computer.

Programming Languages

Programming languages are notations for describing computations to people and to machines.

All the software running on all the computers was written in some programming language.

Before a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called **compilers**.

Language Processors

A compiler is a program that can read a program in one language and translate it into an equivalent program in another language.

Language Processors

A compiler is a program that can read a program in one language and translate it into an equivalent program in another language.

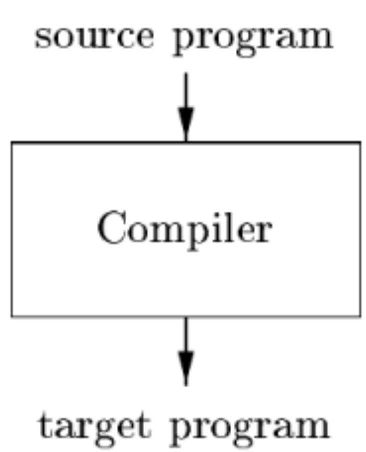


Figure 1.1: A compiler

Language Processors

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

Language Processors

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

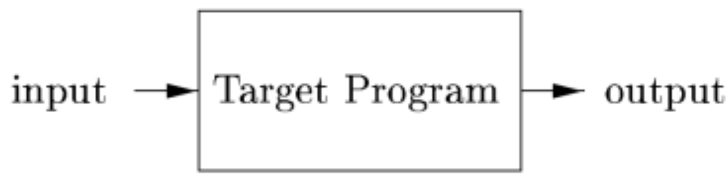


Figure 1.2: Running the target program

Language Processors

An interpreter is another common kind of language processor.

Language Processors

An interpreter is another common kind of language processor.

Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Language Processors

An interpreter is another common kind of language processor.

Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

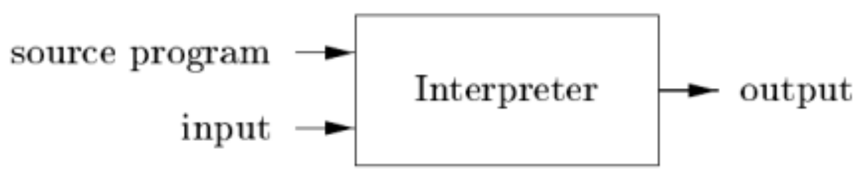


Figure 1.3: An interpreter

Language Processors

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.

Language Processors

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.

Python is an **interpreted**, high-level, general-purpose programming language.

Programming Paradigms

Programming paradigms are a way to classify programming languages based on their features.

Programming Paradigms

Programming paradigms are a way to classify programming languages based on their features.

Languages can be classified into multiple paradigms.

Programming Paradigms

Programming paradigms are a way to classify programming languages based on their features.

Languages can be classified into multiple paradigms.

Common programming paradigms include:

Programming Paradigms

Programming paradigms are a way to classify programming languages based on their features.

Languages can be classified into multiple paradigms.

Common programming paradigms include:

- Imperative
  - Procedural (Fortran, ALGOL, BASIC, Pascal, C)
  - Object-Oriented (Java, C++, C#, ...)

Programming Paradigms

Programming paradigms are a way to classify programming languages based on their features.

Languages can be classified into multiple paradigms.

Common programming paradigms include:

- Imperative
  - Procedural (Fortran, ALGOL, BASIC, Pascal, C)
  - Object-Oriented (Java, C++, C#, ...)
- Functional (Lisp, Scheme, Racket, Erlang, OCaml, Haskell)

Programming Paradigms

Programming paradigms are a way to classify programming languages based on their features.

Languages can be classified into multiple paradigms.

Common programming paradigms include:

- Imperative
  - Procedural (Fortran, ALGOL, BASIC, Pascal, C)
  - Object-Oriented (Java, C++, C#, ...)
- Functional (Lisp, Scheme, Racket, Erlang, OCaml, Haskell)
- Logic (Prolog, Datalog)

Programming Paradigms

Python supports multiple programming paradigms, including:

Programming Paradigms

Python supports multiple programming paradigms, including:

- Object-oriented

Programming Paradigms

Python supports multiple programming paradigms, including:

- Object-oriented
- Imperative

Programming Paradigms

Python supports multiple programming paradigms, including:

- Object-oriented
- Imperative
- Functional

The Static/Dynamic Distinction

If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a static policy or that the issue can be decided at compile time.

The Static/Dynamic Distinction

If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a static policy or that the issue can be decided at compile time.

On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a dynamic policy or to require a decision at run time.

Python is a **dynamically-typed** language.

Static Type Checking

Static typing can find type errors reliably at compile time, which should increase the reliability of the delivered program.

Static Type Checking

Static typing can find type errors reliably at compile time, which should increase the reliability of the delivered program.

Static typing usually results in compiled code that executes faster.

Static Type Checking

Static typing can find type errors reliably at compile time, which should increase the reliability of the delivered program.

Static typing usually results in compiled code that executes faster.

Statically typed languages that lack type inference require that programmers declare the types that a method or function must use.


String -> String -> String -> String
Types are a lie

Static Type Checking

However, a language can be statically typed without requiring type declarations (examples include Haskell, Scala, OCaml), so explicit type declaration is not a necessary requirement for static typing in all languages.

```
Chapter1.hs:46:1: warning: [-Wmissing-signatures]
    Top-level binding with no type signature:
      l1 :: (a -> t1 -> t2) -> (a, t1) -> t2
   |
46 | l1 f t = f (fst t) (snd t)
   | ^^
Ok, one module loaded.
```

Dynamic Type Checking

Dynamic typing performs type checks mostly at run time.

Dynamic Type Checking

Dynamic typing performs type checks mostly at run time.

Dynamic typing may allow compilers to run faster and interpreters to dynamically load new code, because changes to source code in dynamically typed languages may result in less checking to perform and less code to revisit.


DYNAMIC TYPING
STATIC TYPING
DONE!

Strong and Weak Type Systems

In a weakly typed language, the type of a value depends on how it is used.

Strong and Weak Type Systems

In a weakly typed language, the type of a value depends on how it is used.

For example if I can pass a string to the addition operator, in a weakly typed language it will automatically be interpreted as a number or cause an error if the contents of the string cannot be translated into a number.

Strong and Weak Type Systems

In a weakly typed language, the type of a value depends on how it is used.

For example if I can pass a string to the addition operator, in a weakly typed language it will automatically be interpreted as a number or cause an error if the contents of the string cannot be translated into a number.

```
» "image processing" + 2019
← "image processing2019"
```

Strong and Weak Type Systems

In a strongly typed language, a value has a type and that type cannot change.

Strong and Weak Type Systems

In a strongly typed language, a value has a type and that type cannot change.

What you can do to a value depends on the type of the value.

Strong and Weak Type Systems

In a strongly typed language, a value has a type and that type cannot change.

What you can do to a value depends on the type of the value.

```
λ> "image processing" + 2019

<interactive>:1:1: error:
    • No instance for (Num [Char]) arising from a use of ‘+’
    • In the expression: "image processing" + 2019
      In an equation for ‘it’: it = "image processing" + 2019
```

Strong and Weak Type Systems

Strong vs. weak typing is comparable to static vs. dynamic typing.

Strong and Weak Type Systems

Strong vs. weak typing is comparable to static vs. dynamic typing.

In a statically typed language, type checking is performed at compile time; in a dynamically typed language type checking is performed at run time.

Strong and Weak Type Systems

Strong vs. weak typing is comparable to static vs. dynamic typing.

In a statically typed language, type checking is performed at compile time; in a dynamically typed language type checking is performed at run time.

In practice, weakly typed languages are usually dynamically typed.

Back to Python

Use Python for...

›››More

Web Development: Django, Pyramid, Bottle, Tornado, Flask, web2py

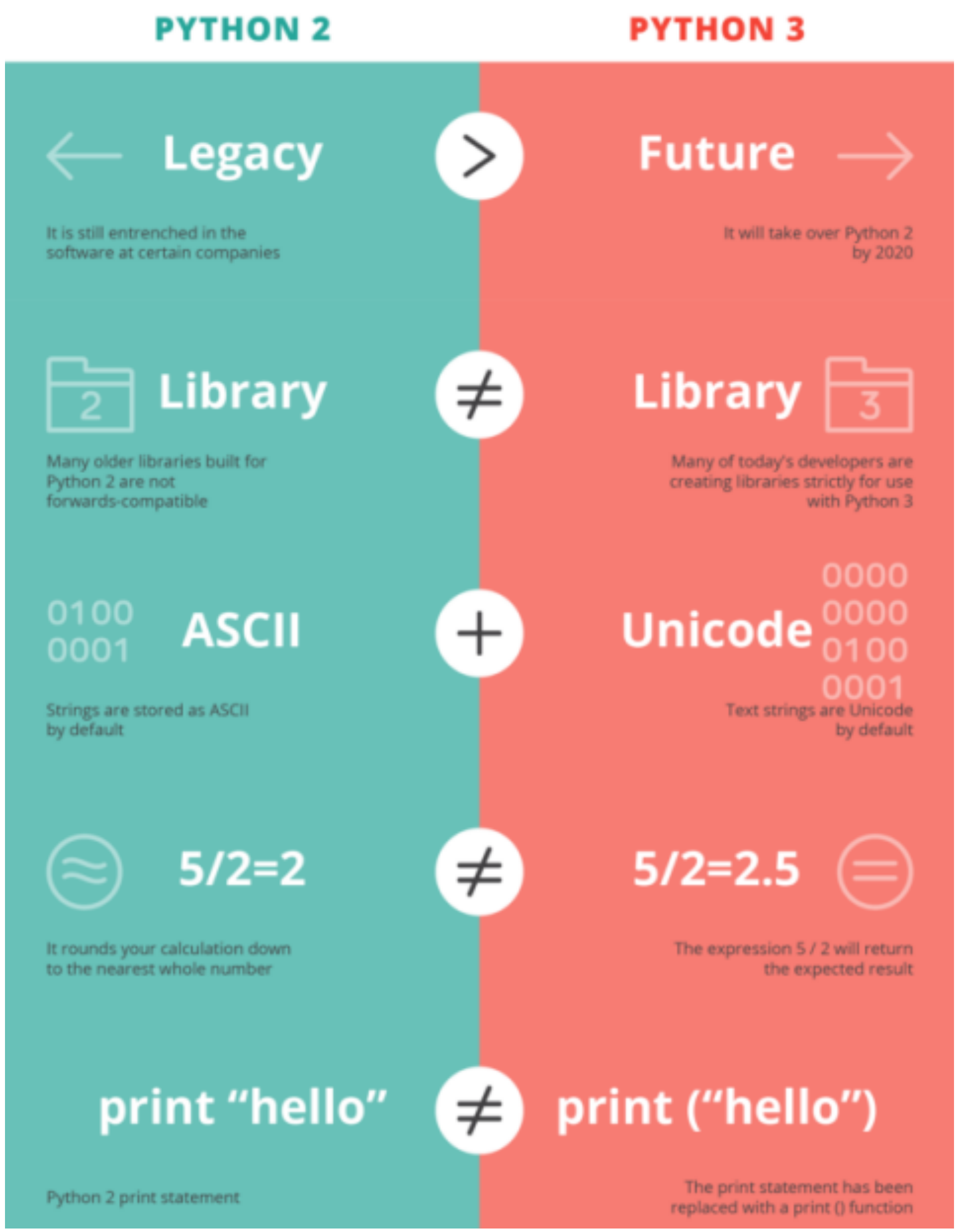
GUI Development: tkInter, PyGObject, PyQt, PySide, Kivy, wxPython

Scientific and Numeric: SciPy, Pandas, IPython

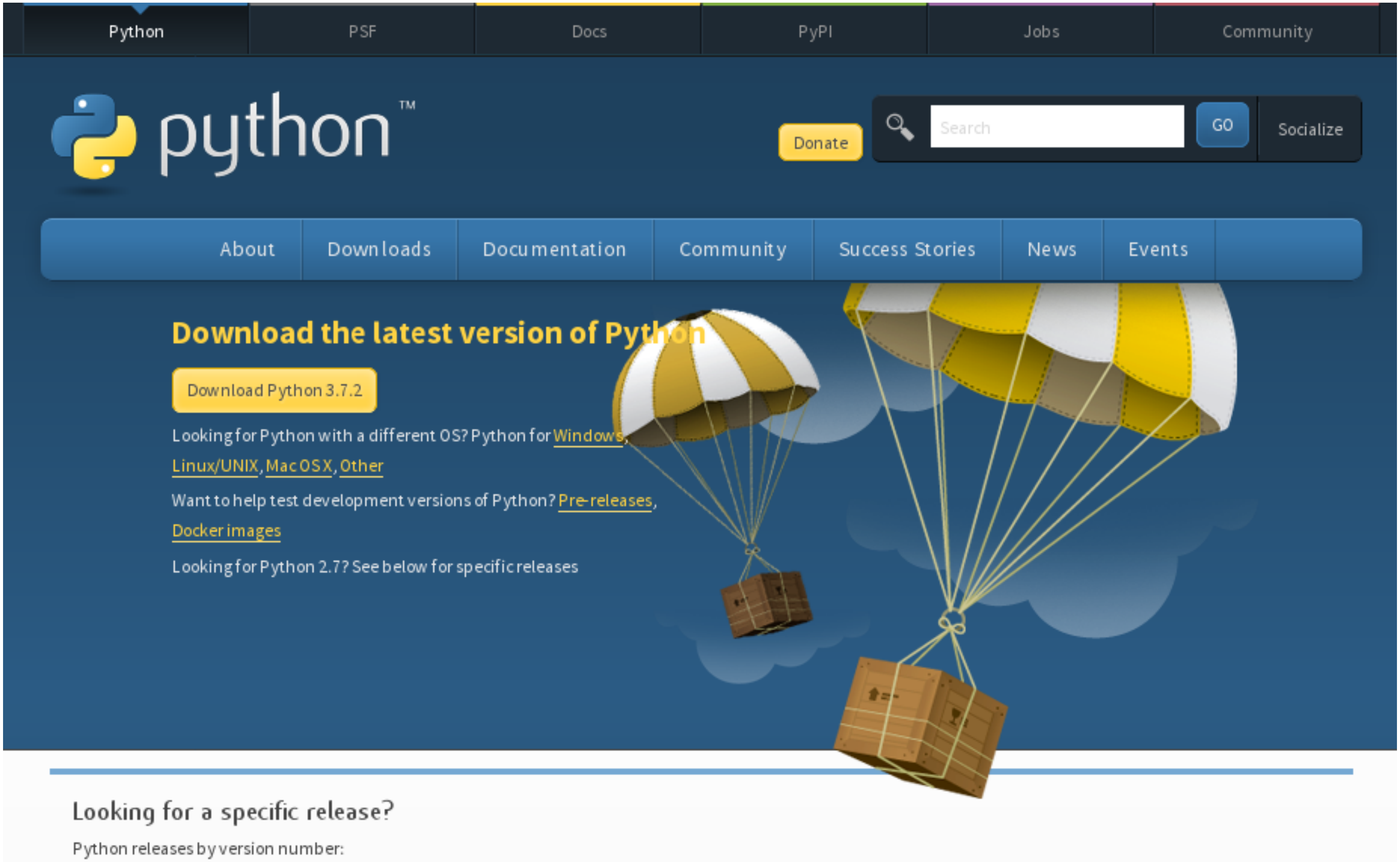
Software Development: Buildbot, Trac, Roundup

System Administration: Ansible, Salt, OpenStack

2 or 3?


PYTHON 2
PYTHON 3
Legacy
Future
It is still entrenched in the software at certain companies
It will take over Python 2 by 2020
Library
Library
2
3
Many older libraries built for Python 2 are not forwards-compatible
Many of today's developers are creating libraries strictly for use with Python 3
0100 0001
ASCII
Unicode
0000 0000 0100 0001
Strings are stored as ASCII by default
Text strings are Unicode by default
5/2=2
5/2=2.5
It rounds your calculation down to the nearest whole number
The expression 5 / 2 will return the expected result
print "hello"
print ("hello")
Python 2 print statement
The print statement has been replaced with a print () function

Download Python



Interpreter

trinket.io/python3

```
deshalb@foundation:~$ python3.5
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Installing Python Modules

(https://packaging.python.org/tutorials/installing-packages)

Installing Python Modules

(https://packaging.python.org/tutorials/installing-packages)

Download **get-pip.py** -> https://bootstrap.pypa.io/get-pip.py

Installing Python Modules

(https://packaging.python.org/tutorials/installing-packages)

Download **get-pip.py** -> https://bootstrap.pypa.io/get-pip.py

Run **python get-pip.py**. This will install or upgrade pip.

Installing Python Modules

(https://packaging.python.org/tutorials/installing-packages)

Download **get-pip.py** -> https://bootstrap.pypa.io/get-pip.py

Run **python get-pip.py**. This will install or upgrade pip.

Additionally, it will install setuptools and wheel if they’re not installed already.

Whetting Your Appetite

```
# Python 3: Fibonacci series up to n
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

fib(1000)
```

Whetting Your Appetite

```python
# Python 3: Fibonacci series up to n
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

fib(1000)
```

```
deshalb@foundation:~/slides/sources$ python3.5 fib.py
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

An Informal Introduction to Python

Python documentation is your friend.

(docs.python.org/3)

An Informal Introduction to Python

Python documentation is your friend.

(docs.python.org/3)

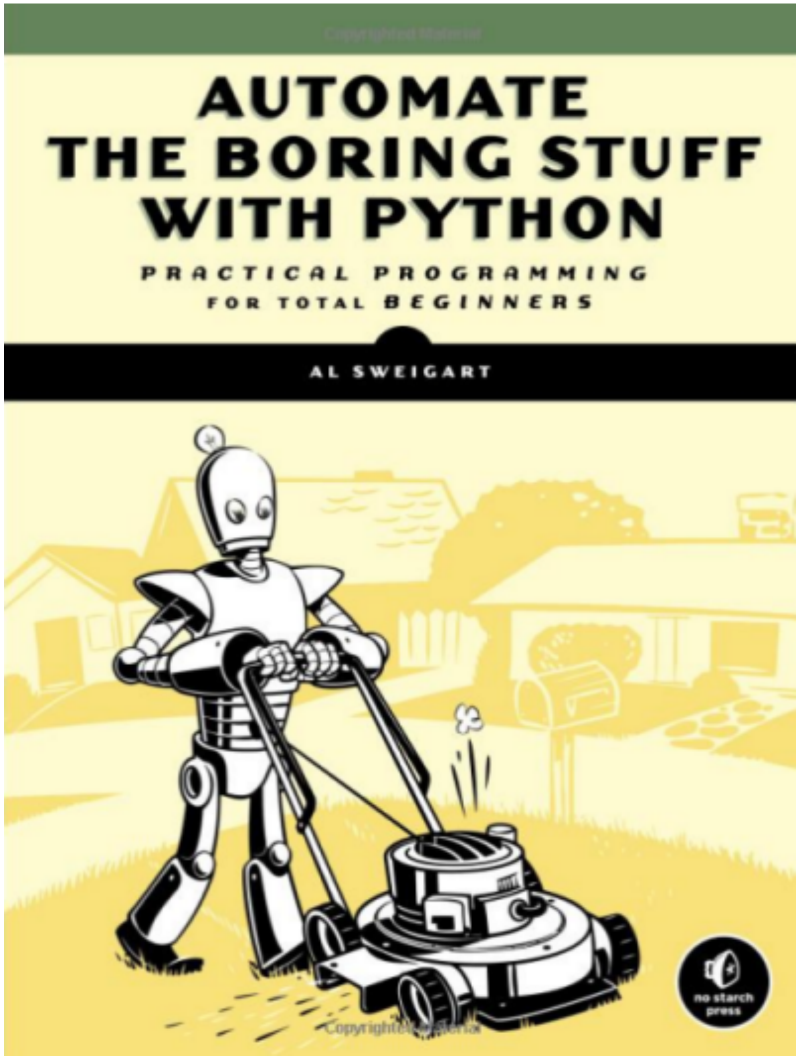
Additional Resources

stanfordpython.com

An Informal Introduction to Python

Automate the Boring Stuff with Python Programming

(automatetheboringstuff.com)



Using Python as a Calculator

Numbers

Using Python as a Calculator

Numbers

The interpreter acts as a simple calculator.

Using Python as a Calculator

Numbers

The interpreter acts as a simple calculator.

You can type an expression at it and it will write the value.

Using Python as a Calculator

Numbers

The interpreter acts as a simple calculator.

You can type an expression at it and it will write the value.

Expression syntax is straightforward: The operators +, -, * and / work just like in most other languages; parentheses (()) can be used for grouping.

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

Using Python as a Calculator

Numbers

The interpreter acts as a simple calculator.

You can type an expression at it and it will write the value.

Expression syntax is straightforward: The operators +, -, * and / work just like in most other languages; parentheses (()) can be used for grouping.

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

The integer numbers (2, 4, 20) have type int, the ones with a fractional part (5.0, 1.6) have type float.

Using Python as a Calculator

You can use **type()** to check the types of your expressions.

Using Python as a Calculator

You can use **type()** to check the types of your expressions.

```
deshalb@foundation:~/slides/sources$ python3.5
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> type(2019)
<class 'int'>
>>> type(2019.0)
<class 'float'>
>>> type("image processing")
<class 'str'>
```

Using Python as a Calculator

Division (/) always returns a float.

Using Python as a Calculator

Division (/) always returns a float.

To do floor division and get an integer result (discarding any fractional result) you can use the // operator

Using Python as a Calculator

Division (/) always returns a float.

To do floor division and get an integer result (discarding any fractional result) you can use the // operator

To calculate the remainder you can use %.

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

Using Python as a Calculator

With Python, it is possible to use the ** operator to calculate powers.

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

Using Python as a Calculator

The equal sign (=) is used to assign a value to a variable.

Using Python as a Calculator

The equal sign (=) is used to assign a value to a variable.

Afterwards, no result is displayed before the next interactive prompt.

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Using Python as a Calculator

If a variable is not “defined” (assigned a value), trying to use it will give you an error.

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Using Python as a Calculator

If a variable is not “defined” (assigned a value), trying to use it will give you an error.

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point

```
>>> 4 * 3.75 - 1
14.0
```

Using Python as a Calculator

In interactive mode, the last printed expression is assigned to the variable _.

Using Python as a Calculator

In interactive mode, the last printed expression is assigned to the variable _.

This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

Using Python as a Calculator

Strings

Using Python as a Calculator

Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways.

Using Python as a Calculator

Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways.

They can be enclosed in single quotes ('...') or double quotes ("...") with the same result. \ can be used to escape quotes.

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

Using Python as a Calculator

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Using Python as a Calculator

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

If you want to concatenate variables or a variable and a literal, use +:

```
>>> prefix + 'thon'
'Python'
```

Using Python as a Calculator

Strings can be indexed (subscripted), with the first character having index 0.

Using Python as a Calculator

Strings can be indexed (subscripted), with the first character having index 0.

There is no separate character type; a character is simply a string of size one.

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

Using Python as a Calculator

In addition to indexing, slicing is also supported.

Using Python as a Calculator

In addition to indexing, slicing is also supported.

While indexing is used to obtain individual characters, slicing allows you to obtain substring.

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Using Python as a Calculator

In addition to indexing, slicing is also supported.

While indexing is used to obtain individual characters, slicing allows you to obtain substring.

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Note how the start is always included, and the end always excluded.

Using Python as a Calculator

In addition to indexing, slicing is also supported.

While indexing is used to obtain individual characters, slicing allows you to obtain substring.

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Note how the start is always included, and the end always excluded.

This makes sure that s[:i] + s[i:] is always equal to s:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Using Python as a Calculator

Attempting to use an index that is too large will result in an error.

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Using Python as a Calculator

Attempting to use an index that is too large will result in an error.

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing.

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Using Python as a Calculator

Python strings cannot be changed — they are immutable.

Using Python as a Calculator

Python strings cannot be changed — they are immutable.

Therefore, assigning to an indexed position in the string results in an error.

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Using Python as a Calculator

If you need a different string, you should create a new one.

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

Using Python as a Calculator

If you need a different string, you should create a new one.

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

The built-in function len() returns the length of a string.

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Using Python as a Calculator

Lists

Python has a number of compound data types, used to group together other values.

Using Python as a Calculator

Lists

Python has a number of compound data types, used to group together other values.

The most versatile is the list, which can be written as a list of comma-separated values 1between square brackets.

Using Python as a Calculator

Lists

Python has a number of compound data types, used to group together other values.

The most versatile is the list, which can be written as a list of comma-separated values 1between square brackets.

Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Using Python as a Calculator

Like strings (and all other built-in sequence type), lists can be indexed and sliced.

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Using Python as a Calculator

Like strings (and all other built-in sequence type), lists can be indexed and sliced.

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

All slice operations return a new list containing the requested elements.

Using Python as a Calculator

Like strings (and all other built-in sequence type), lists can be indexed and sliced.

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

All slice operations return a new list containing the requested elements.

Lists also support operations like concatenation.

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Using Python as a Calculator

Unlike strings, which are immutable, lists are a mutable type. (It is possible to change their content after they've been declared)

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

Using Python as a Calculator

Unlike strings, which are immutable, lists are a mutable type. (It is possible to change their content after they've been declared)

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the append() method.

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Using Python as a Calculator

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely.

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

Using Python as a Calculator

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely.

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

The built-in function len() also applies to lists.

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

Using Python as a Calculator

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

Control Flow Tools

if Statements

Control Flow Tools

if Statements

Perhaps the most well-known statement type is the if statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Control Flow Tools

if Statements

Perhaps the most well-known statement type is the if statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more elif parts, and the else part is optional.

Control Flow Tools

if Statements

Perhaps the most well-known statement type is the if statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more elif parts, and the else part is optional.

The keyword ‘elif’ is short for ‘else if’, and is useful to avoid excessive indentation.

Control Flow Tools

An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages.

Control Flow Tools

An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages.

Example: case in Haskell

```
mbGetBalance :: State Profile (Maybe Int)
mbGetBalance = State $ \t@(Profile email _ balance) ->
  case email of
    Just  _ -> (Just balance, t)
    Nothing -> (Nothing, t)
```

Control Flow Tools

for Statements

Control Flow Tools

for Statements

The for statement in Python differs a bit from what you may be used to in other mainstream languages.

Control Flow Tools

for Statements

The for statement in Python differs a bit from what you may be used to in other mainstream languages.

Rather than giving the user the ability to define both the iteration step and halting condition (as C), Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Control Flow Tools

If you need to modify the sequence you are iterating over while inside the loop, it is recommended that you first make a copy.

Control Flow Tools

If you need to modify the sequence you are iterating over while inside the loop, it is recommended that you first make a copy.

Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient.

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

Control Flow Tools

The range() Function

Control Flow Tools

The range() Function

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy.

Control Flow Tools

The range() Function

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy.

It generates arithmetic progressions.

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Control Flow Tools

The range() Function

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy.

It generates arithmetic progressions.

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

It's also possible to give start, end and step values.

```
range(5, 10)
   5, 6, 7, 8, 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

Control Flow Tools

To iterate over the indices of a sequence, you can combine range() and len() as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Control Flow Tools

To iterate over the indices of a sequence, you can combine range() and len() as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the enumerate() function.

enumerate(*iterable, start=0*)

Return an enumerate object. *iterable* must be a sequence, an iterator, or some other object which supports iteration. The `__next__()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

Control Flow Tools

A strange thing happens if you just print a range:

```
>>> print(range(10))
range(0, 10)
```

Control Flow Tools

A strange thing happens if you just print a range:

```
>>> print(range(10))
range(0, 10)
```

Though the same line in Python 2.x would return:

```
deshalb@foundation:~$ python2.7
Python 2.7.13 (default, Sep 26 2018, 18:42:22)
[GCC 6.3.0 20170516] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Control Flow Tools

A strange thing happens if you just print a range:

```
>>> print(range(10))
range(0, 10)
```

Though the same line in Python 2.x would return:

```
deshalb@foundation:~$ python2.7
Python 2.7.13 (default, Sep 26 2018, 18:42:22)
[GCC 6.3.0 20170516] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

⊙ _ ⊙

That's because in many ways the object returned by range() in Python 3.x behaves as if it is a list, but in fact it isn't.

Control Flow Tools

A strange thing happens if you just print a range:

```
>>> print(range(10))
range(0, 10)
```

Though the same line in Python 2.x would return:

```
deshalb@foundation:~$ python2.7
Python 2.7.13 (default, Sep 26 2018, 18:42:22)
[GCC 6.3.0 20170516] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

⊙ _ ⊙

That's because in many ways the object returned by range() in Python 3.x behaves as if it is a list, but in fact it isn't.

It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

Control Flow Tools

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted.

Control Flow Tools

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Control Flow Tools

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

xrange in Python 2.x behaves in a similar way range in Python 3.x does.

fork(Laziness)

Control Flow Tools

break and continue Statements and else on Loops

Control Flow Tools

break and continue Statements and else on Loops

The break statement, like in C, breaks out of the innermost enclosing for or while loop.

Control Flow Tools

break and continue Statements and else on Loops

The break statement, like in C, breaks out of the innermost enclosing for or while loop.

Loop statements may also have an else clause.

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Control Flow Tools

The continue statement, also borrowed from C, continues with the next iteration of the loop.

Control Flow Tools

The continue statement, also borrowed from C, continues with the next iteration of the loop.

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

Control Flow Tools

pass Statements

Control Flow Tools

pass Statements

The pass statement does nothing.

pass Statements

The pass statement does nothing.

It can be used when a statement is required syntactically but the program requires no action.

Control Flow Tools

pass Statements

The pass statement does nothing.

It can be used when a statement is required syntactically but the program requires no action.

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

```
>>> class MyEmptyClass:
...     pass
...
```

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

Defining Functions

The keyword **def** introduces a function definition.

Defining Functions

The keyword **def** introduces a function definition.

It must be followed by the function name and the parenthesized list of formal parameters.

Defining Functions

The keyword **def** introduces a function definition.

It must be followed by the function name and the parenthesized list of formal parameters.

The statements that form the body of the function start at the next line, and must be indented.

Defining Functions

The keyword **def** introduces a function definition.

It must be followed by the function name and the parenthesized list of formal parameters.

The statements that form the body of the function start at the next line, and must be indented.

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Defining Functions

Coming from other languages, you might object that fib is not a function but a procedure since it doesn't return a value.

Defining Functions

Coming from other languages, you might object that fib is not a function but a procedure since it doesn’t return a value.

In fact, even functions without a return statement do return a value, albeit a rather boring one.

Defining Functions

Coming from other languages, you might object that fib is not a function but a procedure since it doesn’t return a value.

In fact, even functions without a return statement do return a value, albeit a rather boring one.

This value is called **None** (it’s a built-in name).

Defining Functions

Coming from other languages, you might object that fib is not a function but a procedure since it doesn't return a value.

In fact, even functions without a return statement do return a value, albeit a rather boring one.

This value is called **None** (it's a built-in name).

```
>>> fib(0)
>>> print(fib(0))
None
```

Defining Functions

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it.

Defining Functions

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it.

```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Defining Functions

Default Argument Values

Defining Functions

Default Argument Values

The most useful form is to specify a default value for one or more arguments.

Defining Functions

Default Argument Values

The most useful form is to specify a default value for one or more arguments.

This creates a function that can be called with fewer arguments than it is defined to allow.

Defining Functions

Default Argument Values

The most useful form is to specify a default value for one or more arguments.

This creates a function that can be called with fewer arguments than it is defined to allow.

```python
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Defining Functions

Keyword Arguments

Functions can also be called using keyword arguments of the form kwarg=value.

Defining Functions

Keyword Arguments

Functions can also be called using keyword arguments of the form kwarg=value.

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

Defining Functions

When a final formal parameter of the form **name is present, it receives a dictionary containing all keyword arguments.

Defining Functions

When a final formal parameter of the form **name is present, it receives a dictionary containing all keyword arguments.

This may be combined with a formal parameter of the form ***name** which receives a tuple containing the positional arguments beyond the formal parameter list.

Defining Functions

When a final formal parameter of the form **name is present, it receives a dictionary containing all keyword arguments.

This may be combined with a formal parameter of the form ***name** which receives a tuple containing the positional arguments beyond the formal parameter list.

(***name** must occur before ***name**.)

Defining Functions

When a final formal parameter of the form **name is present, it receives a dictionary containing all keyword arguments.

This may be combined with a formal parameter of the form ***name** which receives a tuple containing the positional arguments beyond the formal parameter list.

(***name** must occur before ***name**.)

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Defining Functions

Lambda Expressions

Defining Functions

Lambda Expressions

Small anonymous functions can be created with the lambda keyword.

Defining Functions

Lambda Expressions

Small anonymous functions can be created with the lambda keyword.

This function returns the sum of its two arguments:

Defining Functions

Lambda Expressions

Small anonymous functions can be created with the lambda keyword.

This function returns the sum of its two arguments:

lambda a, b: a+b.

Defining Functions

Lambda Expressions

Small anonymous functions can be created with the lambda keyword.

This function returns the sum of its two arguments:

```
lambda a, b: a+b.
```

Lambda functions can be used wherever function objects are required.

Defining Functions

Lambda Expressions

Small anonymous functions can be created with the lambda keyword.

This function returns the sum of its two arguments:

lambda a, b: a+b.

Lambda functions can be used wherever function objects are required.

They are syntactically restricted to a single expression.

Defining Functions

Lambda Expressions

Small anonymous functions can be created with the lambda keyword.

This function returns the sum of its two arguments:

lambda a, b: a+b.

Lambda functions can be used wherever function objects are required.

They are syntactically restricted to a single expression.

Semantically, they are just syntactic sugar for a normal function definition.

Defining Functions

Lambda Expressions

Small anonymous functions can be created with the lambda keyword.

This function returns the sum of its two arguments:

lambda a, b: a+b.

Lambda functions can be used wherever function objects are required.

They are syntactically restricted to a single expression.

Semantically, they are just syntactic sugar for a normal function definition.

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Coding Style

Coding Style

- Use 4-space indentation, and no tabs.

Coding Style

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don't exceed 79 characters.

Coding Style

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don’t exceed 79 characters.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.

Coding Style

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don’t exceed 79 characters.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.

Coding Style

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don't exceed 79 characters.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: a = f(1, 2) + g(3, 4).

Data Structures

More on Lists

Data Structures

More on Lists

- list.append(x)
  - Add an item to the end of the list.
  - Same as a[len(a):] = [x]

Data Structures

More on Lists

- list.append(x)
    - Add an item to the end of the list.
    - Same as a[len(a):] = [x]
- list.extend(iterable)
    - Extend the list by appending all the items from the iterable.
    - Same as a[len(a):] = iterable.

Data Structures

More on Lists

- list.append(x)
  - Add an item to the end of the list.
  - Same as a[len(a):] = [x]
- list.extend(iterable)
  - Extend the list by appending all the items from the iterable.
  - Same as a[len(a):] = iterable.
- list.insert(i, x)
  - Insert an item at a given position.
  - a.insert(0, x) inserts at the front of the list.
  - a.insert(len(a), x) is equivalent to a.append(x)

Data Structures

- list.remove(x)
  - Remove the first item from the list whose value is equal to x.
  - It raises a ValueError if there is no such item.

Data Structures

- list.remove(x)
  - Remove the first item from the list whose value is equal to x.
  - It raises a ValueError if there is no such item.
- list.pop([i])
  - Remove the item at the given position in the list, and return it.
  - If no index is specified, a.pop() removes and returns the last item in the list.

Data Structures

- list.remove(x)
    - Remove the first item from the list whose value is equal to x.
    - It raises a ValueError if there is no such item.
- list.pop([i])
    - Remove the item at the given position in the list, and return it.
    - If no index is specified, a.pop() removes and returns the last item in the list.
- list.clear()
    - Remove all items from the list. Equivalent to del a[:]

Data Structures

- list.count(x)
    - Return the number of times x appears in the list.

Data Structures

- list.count(x)
  - Return the number of times x appears in the list.
- list.sort(key=None, reverse=False)
  - Sort the items of the list in place.

Data Structures

- list.count(x)
  - Return the number of times x appears in the list.
- list.sort(key=None, reverse=False)
  - Sort the items of the list in place.
- list.reverse()
  - Reverse the elements of the list in place.

Data Structures

- list.count(x)
  - Return the number of times x appears in the list.
- list.sort(key=None, reverse=False)
  - Sort the items of the list in place.
- list.reverse()
  - Reverse the elements of the list in place.
- list.copy()
  - Return a shallow copy of the list.
  - Same as a[:]

Data Structures

An example that uses most of the list methods:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Data Structures

Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”).

Data Structures

Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”).

To add an item to the top of the stack, use append().

Data Structures

Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”).

To add an item to the top of the stack, use append().

To retrieve an item from the top of the stack, use pop() without an explicit index.

Data Structures

Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out").

To add an item to the top of the stack, use append().

To retrieve an item from the top of the stack, use pop() without an explicit index.

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Data Structures

Using Lists as Queues

Data Structures

Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”)

Data Structures

Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”)

However, lists are not efficient for this purpose.

Data Structures

Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”)

However, lists are not efficient for this purpose.

To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends.

Data Structures

Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out")

However, lists are not efficient for this purpose.

To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends.

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

Data Structures

List Comprehensions

Data Structures

List Comprehensions

List comprehensions provide a concise way to create lists.

Data Structures

List Comprehensions

List comprehensions provide a concise way to create lists.

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Data Structures

List Comprehensions

List comprehensions provide a concise way to create lists.

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named x that still exists after the loop completes.

Data Structures

List Comprehensions

List comprehensions provide a concise way to create lists.

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named x that still exists after the loop completes.

We can calculate the list of squares without any side effects using:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

```
squares = [x**2 for x in range(10)]
```

Data Structures

Another example:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Data Structures

Nested List Comprehensions

Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Data Structures

Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

The following list comprehension will transpose rows and columns:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the nested listcomp is evaluated in the context of the `for` that follows it, so this example is equivalent to:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Data Structures

The del statement

Data Structures

The del statement

There is a way to remove an item from a list given its index instead of its value: the del statement.

Data Structures

The del statement

There is a way to remove an item from a list given its index instead of its value: the del statement.

This differs from the pop() method which returns a value.

Data Structures

The del statement

There is a way to remove an item from a list given its index instead of its value: the del statement.

This differs from the pop() method which returns a value.

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

Data Structures

Tuples and Sequences

Data Structures

Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations.

Data Structures

Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations.

They are two examples of sequence data types.

Data Structures

Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations.

They are two examples of sequence data types.

There is also another standard sequence data type: the tuple.

Data Structures

Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations.

They are two examples of sequence data types.

There is also another standard sequence data type: the tuple.

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Data Structures

Though tuples may seem similar to lists, they are often used in different situations and for different purposes.

Data Structures

Though tuples may seem similar to lists, they are often used in different situations and for different purposes.

Tuples are immutable, and usually contain a heterogeneous sequence of elements.

Data Structures

Though tuples may seem similar to lists, they are often used in different situations and for different purposes.

Tuples are immutable, and usually contain a heterogeneous sequence of elements.

Lists are mutable, and their elements are usually homogeneous.

Mutable or Immutable

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Data Structures

A special problem is the construction of tuples containing 0 or 1 items.

Data Structures

A special problem is the construction of tuples containing 0 or 1 items.

The syntax has some extra quirks to accommodate these.

Data Structures

A special problem is the construction of tuples containing 0 or 1 items.

The syntax has some extra quirks to accommodate these.

Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma.

Data Structures

A special problem is the construction of tuples containing 0 or 1 items.

The syntax has some extra quirks to accommodate these.

Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma.

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Data Structures

Sets

Data Structures

Sets

Python also includes a data type for sets.

Data Structures

Sets

Python also includes a data type for sets.

A set is an unordered collection with no duplicate elements.

Data Structures

Sets

Python also includes a data type for sets.

A set is an unordered collection with no duplicate elements.

Basic uses include membership testing and eliminating duplicate entries.

Data Structures

Sets

Python also includes a data type for sets.

A set is an unordered collection with no duplicate elements.

Basic uses include membership testing and eliminating duplicate entries.

Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Data Structures

Curly braces or the set() function can be used to create sets.

Data Structures

Curly braces or the set() function can be used to create sets.

Note: to create an empty set you have to use set(), not {}

Data Structures

Curly braces or the set() function can be used to create sets.

Note: to create an empty set you have to use set(), not {}

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Similarly to list comprehensions, set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

Data Structures

Dictionaries

Data Structures

Dictionaries

Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.

Data Structures

Dictionaries

Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.

Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.

Data Structures

Dictionaries

Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.

Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.

You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

Data Structures

It is best to think of a dictionary as a set of key: value pairs, with the requirement that the keys are unique (within one dictionary).

Data Structures

It is best to think of a dictionary as a set of key: value pairs, with the requirement that the keys are unique (within one dictionary).

A pair of braces creates an empty dictionary: {}.

Data Structures

It is best to think of a dictionary as a set of key: value pairs, with the requirement that the keys are unique (within one dictionary).

A pair of braces creates an empty dictionary: {}.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

The dict() constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Data Structures

Looping Techniques

Data Structures

Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

Data Structures

Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

Data Structures

Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

Data Structures

Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Data Structures

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

Data Structures

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Data Structures

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Or simply:

Data Structures

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Or simply:

```
>>> for x in range(9, 0, -2):
...   print(x)
...
9
7
5
3
1
```

Data Structures

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Or simply:

```
>>> for x in range(9, 0, -2):
...   print(x)
...
9
7
5
3
1
```

Modules

If you quit from the Python interpreter and enter it again, the definitions you have made are lost.

Modules

If you quit from the Python interpreter and enter it again, the definitions you have made are lost.

Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead.

Modules

If you quit from the Python interpreter and enter it again, the definitions you have made are lost.

Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead.

This is known as creating a script.

Modules

A module is a file containing Python definitions and statements.

Modules

A module is a file containing Python definitions and statements.

The file name is the module name with the suffix .py appended.

Modules

A module is a file containing Python definitions and statements.

The file name is the module name with the suffix .py appended.

Within a module, the module's name (as a string) is available as the value of the global variable __name__.

Modules

A module is a file containing Python definitions and statements.

The file name is the module name with the suffix .py appended.

Within a module, the module's name (as a string) is available as the value of the global variable __name__.

For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

Modules

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

What isn't covered

- IO in detail
- Errors and Exceptions
- Classes