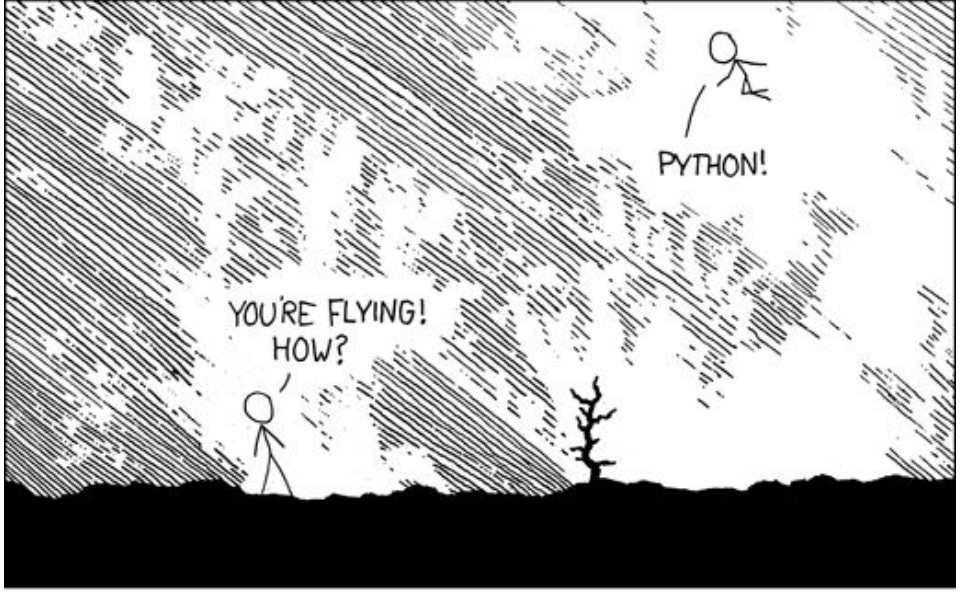
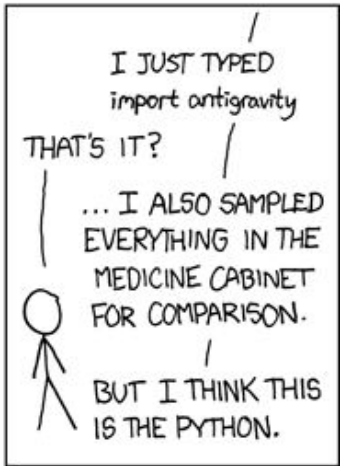

PYTHON!
YOU'RE FLYING! HOW?


I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"


I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?


I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Introduction to Python

Introduction

About Me

My name is Sam Mishra.

I am a Graduate Student at NYU Tandon School of Engineering working on my Masters in Cybersecurity.

I do DevOps at Thesys Technologies

Email: sam640@nyu.edu

Slides: https://github.com/monkeesuit/Intro-To-Python

Introduction

Goals:

> Become familiar with the fundamentals of Python.

- Variables & Data Types
- Control Flow
- Functions
- Object Oriented Programming

> Be comfortable building simple programs

Getting Python Running

Where To Get Python

Mac OSX:

https://www.python.org/downloads/mac-osx/

Windows:

https://www.python.org/downloads/windows/

Linux:

Debian/Ubuntu

```
sudo apt-get install python
sudo apt-get install python3.6
```

Fedora 21

```
sudo yum install python
sudo yum install python3
```

Fedora 22

```
sudo dnf install python
sudo dnf install python3
```

Setting Up Python - Adding Python to PATH

Windows:

> In Search, search for and then select: System (Control Panel)
> Click the **Advanced system settings** link.
> Click **Environment Variables**. In the section **System Variables**, find the PATH environment variable and select it. Click **New**.
> In the **New System Variable** window, specify the path to the Python directory (i.e. C:\Python).
> Click **OK**.

Linux / Mac OSX

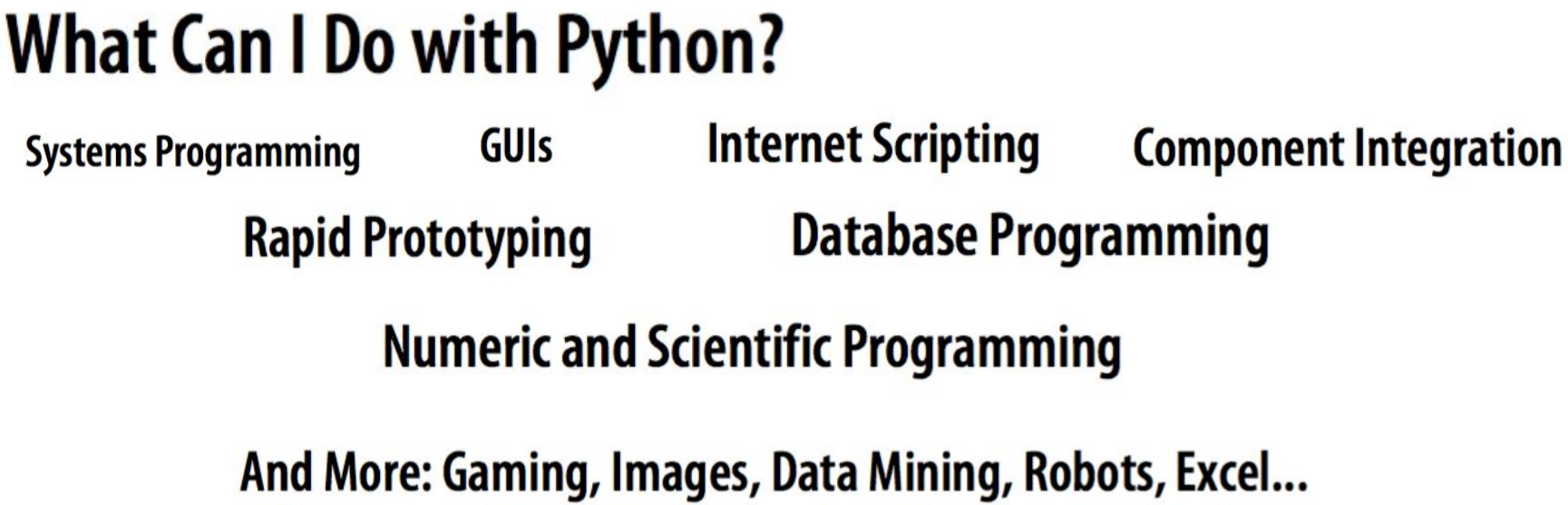
> Depending on which shell you use you'll have to edit either ~/.profile, ~/.zshrc, or ~/.bash_profile using a command line text editor such as vim
> Add this line:

```
export PATH=$PATH:/path/to/python
```

> Save your changes

Python

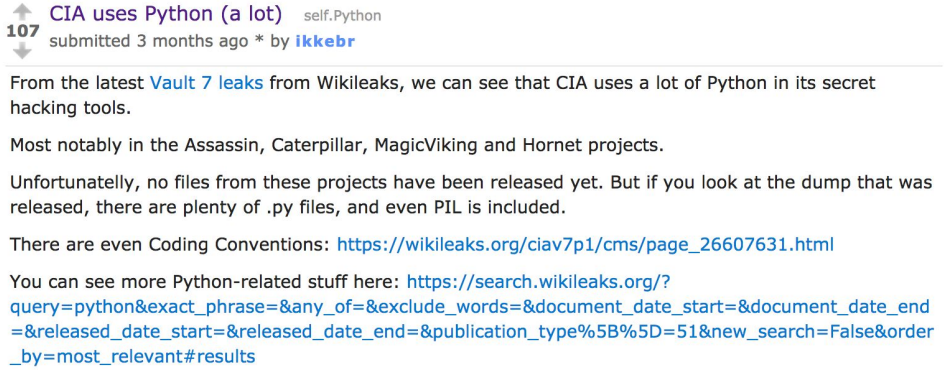
Python is a **high-level programming language** for **general-purpose programming**.
An **interpreted language**, Python has a design philosophy which emphasizes **code readability**.



Python

Python is used by everyone. From the hacking tools of the CIA to the high frequency trading tools used by financial companies.

- *Google* makes extensive use of Python in its web search systems.
- The popular *YouTube* video sharing service is largely written in Python.
- The *Dropbox* storage service codes both its server and desktop client software primarily in Python.
- The *Raspberry Pi* single-board computer promotes Python as its educational language.
- *EVE Online*, a massively multiplayer online game (MMOG) by CCP Games, uses Python broadly.
- The widespread *BitTorrent* peer-to-peer file sharing system began its life as a Python program.



107 CIA uses Python (a lot) self.Python

submitted 3 months ago * by ikkebr

From the latest Vault 7 leaks from Wikileaks, we can see that CIA uses a lot of Python in its secret hacking tools.

Most notably in the Assassin, Caterpillar, MagicViking and Hornet projects.

Unfortunatelly, no files from these projects have been released yet. But if you look at the dump that was released, there are plenty of .py files, and even PIL is included.

There are even Coding Conventions: https://wikileaks.org/ciav7p1/cms/page_26607631.html

You can see more Python-related stuff here: https://search.wikileaks.org/?query=python&exact_phrase=&any_of=&exclude_words=&document_date_start=&document_date_end=&released_date_start=&released_date_end=&publication_type%5B%5D=51&new_search=False&order_by=most_relevant#results

Python

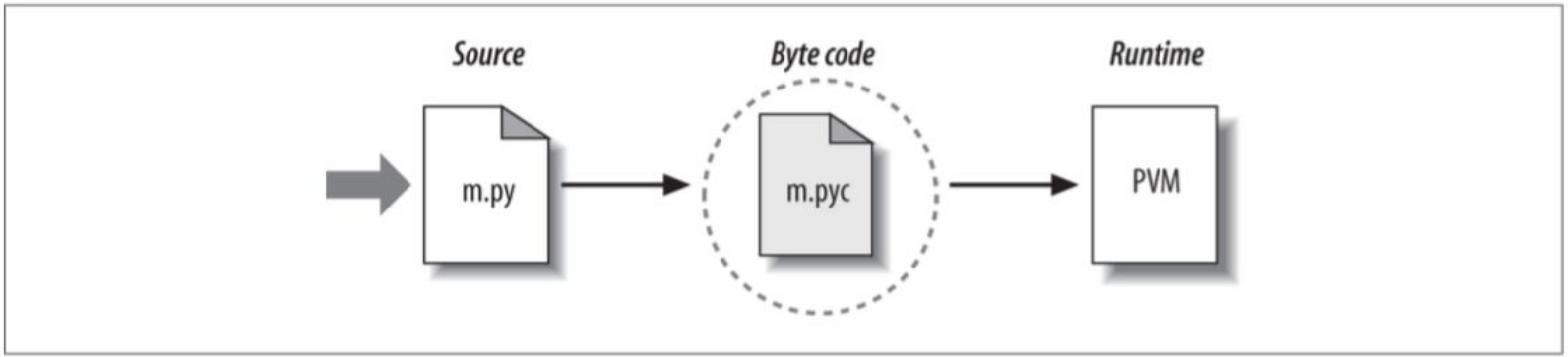
Benefits of Python:

- *Software quality/Developer productivity*
- *Program portability*
- *Support libraries*
- *Component integration*

Downsides of Python:

- its *execution speed* may not always be as fast as that of fully compiled and lower-level languages such as C and C++

Python



- When you execute a program, Python first compiles your source code (the statements in your file) into a format known as byte code.
  - Byte code is a lower-level, platform-independent representation of your source code.
  - Translation is automatic and irrelevant to most Python programs.

- byte code is executed by something generally known as the Python Virtual Machine
  - The PVM is the runtime engine of Python
  - It’s the component that truly runs your scripts.
  - The PVM is a big code loop that iterates through your byte code instructions, one by one, to carry out their operations.

Python

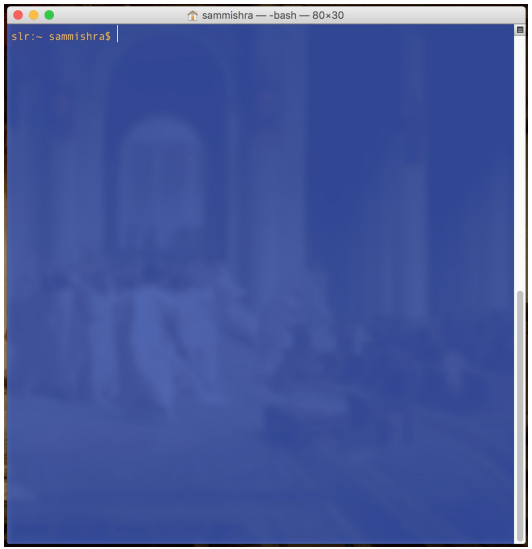
The Zen of Python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

The Command-Line, Paths, Directories & Files

- A **Command-Line Interface**, a.k.a. a shell, is a user-interface that gives access to the operating system's services by typing in commands.
  On Windows: Command Prompt, PowerShell
  On Linux/MacOSX: Terminal, BASH

- This is opposed to a **Graphical User Interface**, which is the more common way of interfacing with an operating system, and uses things like a mouse, icons, windows, menus, etc. to give access to the system's services.



The Command-Line, Paths, Directories & Files

- When using with the Command-Line, folders are referred to as **directories**.

- **Directories** have a hierarchical structure
  - Linux/MacOSX directory structure begins from /.
  - Windows normally starts from C:\.

- In these **directories** are either more **directories** or **files**.
  - You can see the contents of a directory with these commands:
    Linux/MacOSX: *ls*
    Windows*: dir*

- The full description of where a directory or file is is known as the **path**.
  - For example the **path** of file test.txt which is saved on my Desktop would be:
    /Users/sammishra/Desktop/test.txt

- The shell always is ‘in’ one of these directories and can operate on items in this directory directly. This is called the **working directory**.
  - You can find out your current working directory with the command:
    Linux/MacOSX: *pwd*
  - The working directory can be changed with commands:
    Linux/MacOSX: *cd [path to new directory]*
    Windows*: chdir [path to new directory]*

Some Python Syntax

```
In [1]: # set the midpoint
        midpoint = 5

        # make two empty lists
        lower = []; upper = []

        # split the numbers into lower and upper
        for i in range(10):
            if (i < midpoint):
                lower.append(i)
            else:
                upper.append(i)

        print("lower:", lower)
        print("upper:", upper)
```

```
lower: [0, 1, 2, 3, 4]
upper: [5, 6, 7, 8, 9]
```

- Syntax refers to the structure of the language. On the other hand we have semantics - the meaning of the word and symbols
- Comments Are Marked by #
- End-of-Line Terminates a Statement
  - To continue a statement to the next line use the \ or place the statement in parentheses
- A Semicolon Also Terminates a Statement, Allowing Multiple Statements to be Placed on One Line
- Code blocks are noted by indentation
- Whitespace within a line does not matter
- Parentheses Are For Grouping or Calling

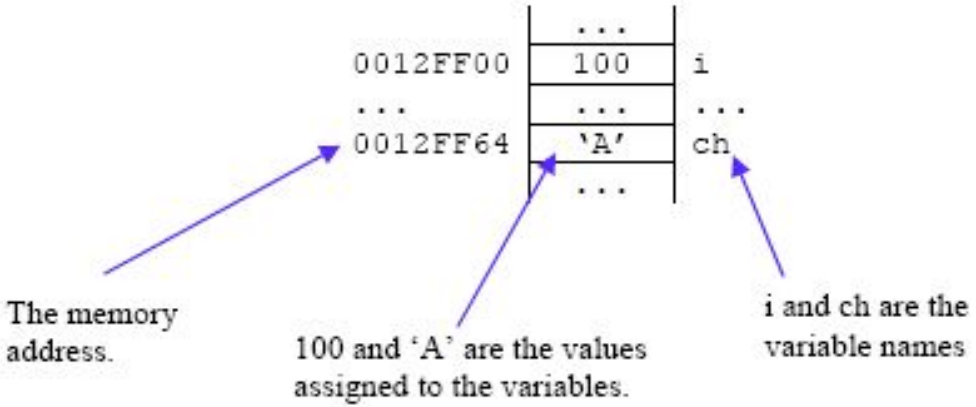
Python Conceptual Hierarchy

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

Variables

In computer programming, a **variable** is a **storage location** *paired with an associated* **symbolic name**/**variable name**/**identifier**, which contains some *information* referred to as a **value**.

The **identifier** is the usual way to reference the stored value



Variables

- To Assign a Variable: Put A **Variable Name** to the Left of the **Equals Sign (=)** and a **Value** to its right

```
# Variable Assignments:
# Assign the integer 100 to variable i
# Assign the string 'A' to variable ch

i = 100
ch = 'A'
```

Variables

Rules for Naming Variables in Python:

[+] **Starts with**: a letter *A* to *Z*, or *a* to *z*, or an underscore _

[+] **Followed by**: zero or more letters, underscores and digits

[+] Punctuation characters [@,$,%] not allowed

[+] Case sensitive - xx and Xx are different

Conventions for Naming Variable in Python:

[+] Readability is very important.

- python_puppet

[+] Descriptive names are very useful.

Variables

- Variables are created when they are first assigned values.
- Variables must be assigned before they can be used in expressions.
- Variables are replaced with their values when used in expressions.

```
>>> a = 3
>>> b = 5
>>> a + 5        # (3+5)
8
>>> a * b        # (3*5)
15
```

Data Types

- In Python values passed to variables are known as **literals** and initialize **objects (data structures)**.
- These literals are linked to one of the core data types found in Python.
- Since Python is an **Object Oriented Programming Language** objects all have **attributes** and **methods** attached to them.
- These attributes and methods are accessed via the dot syntax.

Data Types

Object type	Example literals/creation
Numbers	`1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()`
Strings	`'spam', "Bob's", b'a\x01c', u'sp\xc4m'`
Lists	`[1, [2, 'three'], 4.5], list(range(10))`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}, dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U'), tuple('spam'), namedtuple`
Files	`open('eggs.txt'), open(r'C:\ham.bin', 'wb')`
Sets	`set('abc'), {'a', 'b', 'c'}`
Other core types	Booleans, types, None

Operators

Arithmetic Operations

Operator	Name	Description
`a + b`	Addition	Sum of a and b
`a - b`	Subtraction	Difference of a and b
`a * b`	Multiplication	Product of a and b
`a / b`	True division	Quotient of a and b
`a // b`	Floor division	Quotient of a and b, removing fractional parts
`a % b`	Modulus	Remainder after division of a by b
`a ** b`	Exponentiation	a raised to the power of b
`-a`	Negation	The negative of a
`+a`	Unary plus	a unchanged (rarely used)

Comparison Operations

Operation	Description
`a == b`	a equal to b
`a != b`	a not equal to b
`a < b`	a less than b
`a > b`	a greater than b
`a <= b`	a less than or equal to b
`a >= b`	a greater than or equal to b

Bitwise Operations

Operator	Name	Description
`a & b`	Bitwise AND	Bits defined in both a and b
`a | b`	Bitwise OR	Bits defined in a or b or both
`a ^ b`	Bitwise XOR	Bits defined in a or b but not both
`a << b`	Bit shift left	Shift bits of a left by b units
`a >> b`	Bit shift right	Shift bits of a right by b units
`~a`	Bitwise NOT	Bitwise negation of a

Boolean Operators

Operation	Result
`x or y`	if *x* is false, then *y*, else *x*
`x and y`	if *x* is false, then *x*, else *y*
`not x`	if *x* is false, then `True`, else `False`

Identity and Membership Operators

Operator	Description
`a is b`	True if a and b are identical objects
`a is not b`	True if a and b are not identical objects
`a in b`	True if a is a member of b
`a not in b`	True if a is not a member of b

Python Conceptual Hierarchy

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

Expressions

- A combination of **objects** and **operators** that **computes a value** when executed
    - For instance:
        to add two numbers X and Y you would say X + Y,

- Rules:

Mixed operators follow operator precedence

Parentheses group subexpressions

Mixed types are converted up

Operator overloading and polymorphism

Expressions

Operators	Description
`yield x`	Generator function send protocol
`lambda args: expression`	Anonymous function generation
`x if y else z`	Ternary selection (x is evaluated only if y is true)
`x or y`	Logical OR (y is evaluated only if x is false)
`x and y`	Logical AND (y is evaluated only if x is true)
`not x`	Logical negation
`x in y,x not in y`	Membership (iterables, sets)
`x is y,x is not y`	Object identity tests
`x < y,x <= y,x > y,x >= y`	Magnitude comparison, set subset and superset;
`x == y,x != y`	Value equality operators
`x | y`	Bitwise OR, set union
`x ^ y`	Bitwise XOR, set symmetric difference
`x & y`	Bitwise AND, set intersection
`x << y,x >> y`	Shift x left or right by y bits
`x + y`	Addition, concatenation;
`x - y`	Subtraction, set difference
`x * y`	Multiplication, repetition;
`x % y`	Remainder, format;
`x / y,x // y`	Division: true and floor
`-x,+x`	Negation, identity
`~x`	Bitwise NOT (inversion)
`x ** y`	Power (exponentiation)
`x[i]`	Indexing (sequence, mapping, others)
`x[i:j:k]`	Slicing
`x(...)`	Call (function, method, class, other callable)
`x.attr`	Attribute reference
`(...)`	Tuple, expression, generator expression
`[...]`	List, list comprehension
`{...}`	Dictionary, set, set and dictionary comprehensions

Data Types

Mutability vs. Immutability

- In general, data types in Python can be distinguished based on whether objects of the type are **mutable** or **immutable**.
- Immutable types **cannot be changed after they are created**.
- Mutable types support methods that **change the object in place**

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 Types

Mutability vs. Immutability

```
x = something # immutable type
y = x
print x
# some statement that operates on y
print x # prints the same thing

x = something # mutable type
y = x
print x
# some statement that operates on y
print x # might print something different
```

Data Types

Mutability vs. Immutability

Mutable List:

```
>>> x = ['Alice']
>>> y = x
>>> print(x, y)
['Alice'] ['Alice']
>>> y.append('Bob')
>>> print(x, y)
['Alice', 'Bob'] ['Alice', 'Bob']
```

Immutable String:

```
>>> x = 'blue'
>>> y = x
>>> print(x, y)
blue blue
>>> y += 'cheese'
>>> print(x, y)
blue bluecheese
```

Data Types - Numeric Types

There are three distinct numeric types:
integers, *floating point numbers*, and *complex numbers (not going to cover)*.

- **int (signed integers)**
  - Positive or negative whole numbers.
  - Immutable.

- **float (floating point real values)** :
  - Represent real numbers.
  - Written with a decimal point dividing the integer and fractional parts.
  - May also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 250).
  - Immutable.
  - ★ Float can do some unexpected things.
    Read about it here:
    https://docs.python.org/3/tutorial/floatingpoint.html

Data Types - Numeric Types

Arithmetic Operations

Operator	Name	Description
a + b	Addition	Sum of a and b
a - b	Subtraction	Difference of a and b
a * b	Multiplication	Product of a and b
a / b	True division	Quotient of a and b
a // b	Floor division	Quotient of a and b, removing fractional parts
a % b	Modulus	Remainder after division of a by b
a ** b	Exponentiation	a raised to the power of b
-a	Negation	The negative of a
+a	Unary plus	a unchanged (rarely used)

Comparison Operations

Operation	Description
a == b	a equal to b
a != b	a not equal to b
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

Data Types - Numeric Types

Integers in Python 3.X:

- there is only integer (as opposed to Python 2.X), which automatically supports unlimited precision
  - i.e. There is no limit for the length of integer literals apart from what can be stored in available memory.

- Integers may be coded in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2),
  - Hexadecimals start with a leading 0x or 0X, followed by a string of hexadecimal digits (0–9 and A–F). Hex digits may be coded in lower- or uppercase.
  - Octal literals start with a leading 0o or 0O (zero and lower- or uppercase letter o), followed by a string of digits (0–7).
  - Binary literals begin with a leading 0b or 0B, followed by binary digits (0–1).

```
>>> x = 0x0A
>>> x
10
>>> y = 0b1010
>>> y
10
```

```
>>> hex(10)
'0xa'
>>> bin(10)
'0b1010'
>>> int(0b1010)
10
>>> int(0x0a)
10
```

Data Types - Sets

- an **unordered collection** of **unique** and **immutable objects** that supports **operations corresponding to mathematical set theory**.
- The set type is **mutable** — the contents can be changed using methods like add() and remove().
- **Create Sets:** Curly braces - '{}' - or the set() function can be used.
    - Note: to create an empty set you have to use `set()`, not `{}`;
- **Common Uses**: membership testing, removing duplicates from a sequence, computing mathematical operations such as intersection, union, difference, and symmetric difference.

Data Types - Sets

len(s)
Return the number of elements in set *s* (cardinality of *s*).

x in s
Test *x* for membership in *s*.

x not in s
Test *x* for non-membership in *s*.

isdisjoint(*other*)
Return `True` if the set has no elements in common with *other*. Sets are disjoint if and only if their intersection is the empty set.

issubset(*other*)
set <= other
Test whether every element in the set is in *other*.

set < other
Test whether the set is a proper subset of *other*, that is, `set <= other and set != other`.

issuperset(*other*)
set >= other
Test whether every element in *other* is in the set.

set > other
Test whether the set is a proper superset of *other*, that is, `set >= other and set != other`.

union(**others*)
set | other | ...
Return a new set with elements from the set and all others.

intersection(**others*)
set & other & ...
Return a new set with elements common to the set and all others.

difference(**others*)
set - other - ...
Return a new set with elements in the set that are not in the others.

symmetric_difference(*other*)
set ^ other
Return a new set with elements in either the set or *other* but not both.

copy()
Return a new set with a shallow copy of *s*.

update(**others*)
set |= other | ...
Update the set, adding elements from all others.

intersection_update(**others*)
set &= other & ...
Update the set, keeping only elements found in it and all others.

difference_update(**others*)
set -= other | ...
Update the set, removing elements found in others.

symmetric_difference_update(*other*)
set ^= other
Update the set, keeping only elements found in either set, but not in both.

add(*elem*)
Add element *elem* to the set.

remove(*elem*)
Remove element *elem* from the set. Raises `KeyError` if *elem* is not contained in the set.

discard(*elem*)
Remove element *elem* from the set if it is present.

pop()
Remove and return an arbitrary element from the set. Raises `KeyError` if the set is empty.

clear()
Remove all elements from the set.

Data Types - Sets

```
>>> 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 Types - Sets

Why Use Sets?

- Filter duplicates out of a collection
- Isolate differences
- Order-neutral equality
- Convenient when dealing with large data sets i.e. databases

Data Types - Strings

- Textual data in Python is handled with **str** objects, or **strings**.
- Strings are **immutable sequences** of Unicode code points.
- Strings can be used to represent just about anything that can be encoded as text or bytes.
  - Words, text files, bits being transmitted over the internet, etc.
- No character terminates a string in Python.

Data Types - Strings

String literals are written in a variety of ways:

- Single quotes: `'allows embedded "double" quotes'`
- Double quotes: `"allows embedded 'single' quotes"`.
- Triple quoted: `'''Three single quotes'''`, `"""Three double quotes"""`
    - Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

Data Types - Strings

- Backslashes are used to introduce special character codings known as escape sequences.
- The character \, and one or more characters following it in the string literal are replaced with a single character in the resulting string object.

Escape Sequence	Meaning
newline	Ignored
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
ooo	ASCII character with octal value *ooo*
\x*hh*...	ASCII character with hex value *hh*...

Data Types - Strings

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then –

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1

Data Types - Strings

Indexing and Slicing a String:

- Strings are ordered collections of characters.
  - We can access their components by position.
  - Offsets start at 0 and end at one less that the length of the string.

Data Types - Strings

Indexing (`S[i]`) fetches components at offsets:

- The first item is at offset 0.
- Negative indexes mean to count backward from the end or right.
- `S[0]` fetches the first item.
- `S[-2]` fetches the second item from the end (like `S[len(S)-2]`).

Slicing (`S[i:j]`) extracts contiguous sections of sequences:

- The upper bound is noninclusive.
- Slice boundaries default to 0 and the sequence length, if omitted.
- `S[1:3]` fetches items at offsets 1 up to but not including 3.
- `S[1:]` fetches items at offset 1 through the end (the sequence length).
- `S[:3]` fetches items at offset 0 up to but not including 3.
- `S[:-1]` fetches items at offset 0 up to but not including the last item.
- `S[:]` fetches items at offsets 0 through the end—making a top-level copy of `S`.

Extended slicing (`S[i:j:k]`) accepts a step (or stride) `k`, which defaults to +1:

- Allows for skipping items and reversing order—see the next section.

Data Types - Strings

list(s)	returns string s as a list of characters
s.lower(), s.upper()	returns the lowercase or uppercase version of the string
s.strip()	returns a string with whitespace removed from the start and end
s.isalpha() / s.isdigit() / s.isspace()	tests if all the string chars are in the various character classes
s.startswith('other'), s.endswith('other')	tests if the string starts or ends with the given other string
s.find('other')	searches for the given other string within s, and returns the first index where it begins or -1 if not found
s.replace('old', 'new')	returns a string where all occurrences of 'old' have been replaced by 'new'
s.split('delim')	returns a list of substrings separated by the given delimiter.
s.join(list)	opposite of split(), joins the elements in the given list together using the string as the delimiter.

Data Types - Strings

String Formatting

- string formatting allows us to perform multiple type-specific substitutions on a string in a single step.
- It's never strictly required, but it can be convenient, especially when formatting text to be displayed to a program's users.
- 2 types of ways to format:
  - Using an expression
  - Using a method -> We will only talk about this way

str.**format**(**args*, ***kwargs*)

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

Input/Output

- **standard streams** are input and output communication channels between a computer program and its environment of execution.
- The three standard streams are **standard input** (**stdin**), **standard output** (**stdout**) and **standard error** (**stderr**).
- Originally I/O happened via a physically connected system - input via keyboard, output via monitor - standard streams **abstract** this.
    - When a command is executed via a shell, the streams are typically connected to the text terminal on which the shell is running.

input([*prompt*])

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that.

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

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as keyword arguments.

Input/Output

```
>>> x = input('When you see this type something > ')
When you see this type something > which Witch is which
>>> print(x)
which Witch is which
```

Data Types - Sequence Types

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Data Types - Sequence Types

Immutable Sequence Types

The only operation that immutable sequence types generally implement that is not also implemented by mutable sequence types is support for the `hash()` built-in.

Mutable Sequence Types

In the table *s* is an instance of a mutable sequence type, *t* is any iterable object and *x* is an arbitrary object that meets any type and value restrictions imposed by *s*.

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from `s` (same as `del s[:]`)
`s.copy()`	creates a shallow copy of `s` (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop([i])`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i] == x`
`s.reverse()`	reverses the items of *s* in place

Data Types - Lists

- Lists are Python's most flexible **ordered** collection object type.

- Lists can contain any sort of object: number, strings, and even other lists.
  - Typically used to store collections of homogeneous items.
  - This is because we often like to loop over the elements of a list and apply a common function to each element.

- Lists are **mutable sequences.** Lists may be changed in place using:
  - Indexing and Slicing
  - List Methods
  - Deletion Statements

Data Types - Lists

- Ordered Collections of Arbitrary Objects
  - Lists are places to collect other objects so that you can treat them as groups.
  - Lists maintain left-to-right positional ordering among the items they contain (starting from index 0).
- Accessed by Offset
  - You can fetch an object from the list by indexing the list on the object's offset.
    i.e - object = list[index]
- Variable-length, Heterogeneous, and Arbitrarily Nestable
  - Lists can grow and shrink in place.
  - Can contain any type of objects including lists. We can have lists of lists.
- Of the Category "Mutable Sequence"
  - Lists support all the sequence operations we saw with strings.
  - These operations do not return a new list, but instead changes the object in place.
  - Lists also support the operations associated with mutable sequence objects.
- Arrays of Object References
  - Python lists are arrays not linked structures.
  - Reminiscent of a C array of pointers.

Data Types - Lists

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: `[]`
- Using square brackets, separating items with commas: `[a]`, `[a, b, c]`
- Using a list comprehension: `[x for x in iterable]`
- Using the type constructor: `list()` or `list(iterable)`

Data Types - Lists

Lists implement the sequence operations.

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Lists also implement mutable sequence operations.

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from `s` (same as `del s[:]`)
`s.copy()`	creates a shallow copy of `s` (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop([i])`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i] == x`
`s.reverse()`	reverses the items of *s* in place

Lists also provide the following additional method:

s.sort(*key=None*, *reverse=None*)

- This method sorts the list in place, using only `<` comparisons between items.

Data Types - Lists

Common Mistake:

- List Methods change the associated list object **IN PLACE**.
- No object is returned as a result of calling these methods.

Don't do this:

```
>>> L = ['obj1', 'obj2', 'obj3']
>>> L
['obj1', 'obj2', 'obj3']
>>> L = L.append('obj4')
>>> L
>>> print(L)
None
```

Correct Way:

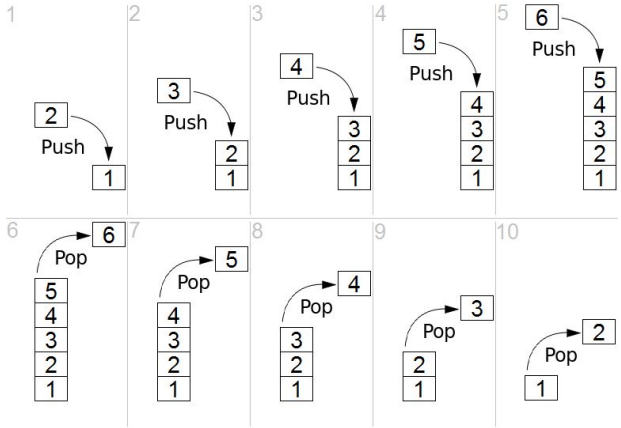
```
>>> L = ['obj1', 'obj2', 'obj3']
>>> L.append('obj4')
>>> L
['obj1', 'obj2', 'obj3', 'obj4']
```

Data Types - Lists

STACK: abstract data type that serves as a collection of elements

- Two principal operations:
  - ***push***, which adds an element to the collection,
  - ***pop***, which removes the most recently added element that was not yet removed.

- The order in which elements come off a stack gives rise to its alternative name, LIFO (last in, first out).



Data Types - Lists

```
>>> L = []
>>> L.append('alice')
>>> L
['alice']
>>> L.append('bob')
>>> L
['alice', 'bob']
>>> L.append('sally')
>>> L
['alice', 'bob', 'sally']
```

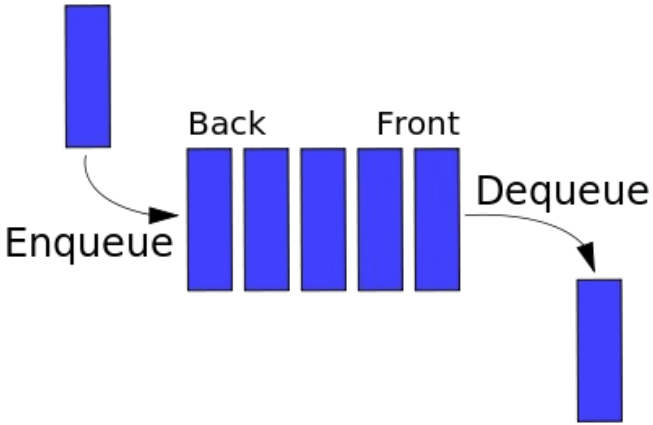
```
>>> L.pop()
'sally'
>>> L
['alice', 'bob']
>>> L.pop()
'bob'
>>> L
['alice']
>>> L.pop()
'alice'
>>> L
[]
```

Data Types - Lists

QUEUE: abstract data type that serves as a collection in which the entities in the collection are kept in order.

- The operations on the collection are:
  - ***enqueue***: the addition of entities to the rear terminal position.
  - ***dequeue***: removal of entities from the front terminal position.

- This makes the queue First-In-First-Out (FIFO)



Data Types - Lists

```
>>> L = []
>>> L.insert(0, 'joe')
>>> L
['joe']
>>> L.insert(0, 'bob')
>>> L
['bob', 'joe']
>>> L.insert(0, 'sally')
>>> L
['sally', 'bob', 'joe']
```

```
>>> L.pop()
'joe'
>>> L
['sally', 'bob']
>>> L.pop()
'bob'
>>> L
['sally']
>>> L.pop()
'sally'
>>> L
[]
```

Data Types - Tuples

- Tuples construct simple groups of objects.
- Tuples are similar to lists except that they are **immutable sequences** - cannot be changed in place.
  - Tuple immutability only applies to the top level of the tuple itself not to its contents.
    i.e. - We can have a tuple that contains a list. This list is still mutable.
- Tuple are typically used over a list when:
  - There is a need for integrity - no other variable can point to our immutable tuple object.
  - A list cannot be used - dictionary keys.

Data Types - Tuples

- Ordered collections of arbitrary objects
  - Like strings and lists, tuples are positionally ordered collections of objects.
  - Like lists, they can embed any kind of object.
- Accessed by offset
  - Like strings and lists, items in a tuple are accessed by offset
  - They support all the offset-based access operations, such as indexing and slicing.
- Of the category “immutable sequence”
  - Like strings and lists, tuples are sequences
  - However, like strings, tuples are immutable; they don’t support any of the in-place change operations applied to lists.
- Fixed-length, heterogeneous, and arbitrarily nestable
  - Because tuples are immutable, you cannot change the size of a tuple.
  - Tuples can hold any type of object.
- Arrays of object references
  - Like lists, tuples are best thought of as object reference arrays.

Data Types - Tuples

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: `()`
- Using a trailing comma for a singleton tuple: `a,` or `(a,)`
- Separating items with commas: `a, b, c` or `(a, b, c)`
- Using the tuple() built-in: `tuple()` or `tuple(iterable)`

Data Types - Tuples

Tuples implement all of the common sequence operations.

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Data Types - Dictionary

- Dictionaries are **unordered collections**.
- Dictionaries **map: immutable keys** to **any data type**.
- Dictionaries are used when item keys are more meaningful than item position.
  - Items are stored and fetched by a key instead of an index.
- Dictionaries are **mutable** objects.
  - You can change, expand and shrink them in place.

Data Types - Dictionary

- Accessed by key, not offset position
  - Dictionaries associate a set of values with keys.
  - You can fetch an item out of a dictionary using the key under which you originally stored it.
- Unordered collections of arbitrary objects
  - Unlike in a list, items stored in a dictionary aren't kept in any particular order.
- Variable-length, heterogeneous, and arbitrarily nestable
  - Like lists, dictionaries can grow and shrink in place.
  - They can contain objects of any type.
  - Each key can have just one associated value, but that value can be a collection type.
- Of the category "mutable mapping"
  - Dictionaries don't support the sequence operations that work on strings and lists.
- Tables of object references (hash tables)
  - Dictionaries are implemented as hash tables (data structures that support very fast retrieval), which start small and grow on demand.
  - Like lists, dictionaries store object references.

Data Types - Dictionary

The following examples all return a dictionary equal to {"one": 1, "two": 2,"three": 3}:

```
>>> a = dict(one=1, two=2, three=3)

>>> b = {'one': 1, 'two': 2, 'three': 3}

>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))

>>> d = dict([('two', 2), ('one', 1), ('three', 3)])

>>> e = dict({'three': 3, 'one': 1, 'two': 2})

>>> a == b == c == d == e
True
```

Data Types - Dictionary

- Dictionaries compare **equal** (‘==’), if and only if, they have the same **key: value** pairs.
- Operations that depend on fixed positional order (sequence operations) don’t work with dictionaries.
  - Concatenation, Slicing, etc.
  - Order comparisons (‘<’, ‘<=’, ‘>=’, ‘>’) raise TypeError.
- Assigning to new indexes adds entries.
  - Keys can be created during dictionary creation or added later on
- Keys do not have to be strings. Any immutable object (numeric types, strings, tuples) can be a key.

Data Types - Dictionary

`len(d)`
Return the number of items in the dictionary *d*.

`d[key]`
Return the item of *d* with key *key*. Raises a `KeyError` if *key* is not in the map.

If a subclass of dict defines a method `__missing__()` and *key* is not present, the `d[key]` operation calls that method with the key *key* as argument. The `d[key]` operation then returns or raises whatever is returned or raised by the `__missing__(key)` call. No other operations or methods invoke `__missing__()`. If `__missing__()` is not defined, `KeyError` is raised. `__missing__()` must be a method; it cannot be an instance variable:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of `collections.Counter`. A different `__missing__` method is used by `collections.defaultdict`.

`d[key] = value`
Set `d[key]` to *value*.

`del d[key]`
Remove `d[key]` from *d*. Raises a `KeyError` if *key* is not in the map.

`key in d`
Return `True` if *d* has a key *key*, else `False`.

`key not in d`
Equivalent to `not key in d`.

`iter(d)`
Return an iterator over the keys of the dictionary. This is a shortcut for `iter(d.keys())`.

Data Types - Dictionary

dict.**fromkeys**(*seq*[, *value*])

Create a new dictionary with keys from *seq* and values set to *value*.

`fromkeys()` is a class method that returns a new dictionary. *value* defaults to `None`.

dict.**get**(*key*[, *default*])

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

dict.**items**()

Return a new view of the dictionary's items (`(key, value)` pairs). See the documentation of view objects.

dict.**keys**()

Return a new view of the dictionary's keys. See the documentation of view objects.

dict.**pop**(*key*[, *default*])

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

dict.**popitem**()

Remove and return an arbitrary `(key, value)` pair from the dictionary.

`popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling `popitem()` raises a `KeyError`.

dict.**setdefault**(*key*[, *default*])

If *key* is in the dictionary, return its value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to `None`.

dict.**update**([*other*])

Update the dictionary with the key/value pairs from *other*, overwriting existing keys. Return `None`.

`update()` accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

dict.**values**()

Return a new view of the dictionary's values. See the documentation of view objects.

Python Conceptual Hierarchy

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

Statements

- **Expressions** process **objects** and are embedded in **statements**.
- **Statements** code the larger logic of a program's operation
  - they use and direct expressions to process the objects
- Statements are where objects spring into existence
  - e.g., in expressions within assignment statements
    X = Y + Z

Statements

Statement	Role	Example
Assignment	Creating references	`a, b = 'good', 'bad'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
`print` calls	Printing objects	`print('The Killer', joke)`
`if/elif/else`	Selecting actions	`if "python" in text:` `  print(text)`
`for/else`	Iteration	`for x in mylist:` `  print(x)`
`while/else`	General loops	`while X > Y:` `  print('hello')`
`pass`	Empty placeholder	`while True:` `  pass`
`break`	Loop exit	`while True:` `  if exittest(): break`
`continue`	Loop continue	`while True:` `  if skiptest(): continue`
`def`	Functions and methods	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
`return`	Functions results	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
`yield`	Generator functions	`def gen(n):` `  for i in n: yield i*2`

`global`	Namespaces	`x = 'old'`
		`def function():`
		`  global x, y; x = 'new'`

`nonlocal`	Namespaces (3.X)	`def outer():`
		`  x = 'old'`
		`  def function():`
		`  nonlocal x; x = 'new'`

`import`	Module access	`import sys`
`from`	Attribute access	`from sys import stdin`

`class`	Building objects	`class Subclass(Superclass):`
		`  staticData = []`
		`  def method(self): pass`

`try/except/finally`	Catching exceptions	`try:`
		`  action()`
		`except:`
		`  print('action error')`

`raise`	Triggering exceptions	`raise EndSearch(location)`
`assert`	Debugging checks	`assert X > Y, 'X too small'`
`with/as`	Context managers (3.X, 2.6+)	`with open('data') as myfile:` `  process(myfile)`

`del`	Deleting references	`del data[k]`
		`del data[i:j]`
		`del obj.attr`
		`del variable`

Statements

the colon character (:)

- All Python **compound statements**—statements that have other statements nested inside them—follow the same general pattern of a **header line terminated in a colon**, followed by a **nested (indented) block of code,** like this:

```
Header line:
    Nested statement block
```

- **The colon is required**

Statements

- You don’t need to terminate statements with semicolons in Python the way you do in C-like languages:

- In Python, the general rule is that the end of a line automatically terminates the statement that appears on that line.

- There are some ways to work around this rule

Statements

- It is possible to squeeze more than one statement onto a single line by separating them with semicolons (‘;’) known as statement separators:

```
a = 1; b = 2; print(a + b)          # Three statements on one line
```

Statements

- You can make a single statement span across multiple lines.
- You simply have to enclose part of your statement in a bracketed pair
  - parentheses (()), square brackets ([]), or curly braces ({}).
- Any code enclosed in these constructs can cross multiple lines: your statement doesn't end until Python reaches the line containing the closing part of the pair. For instance, to continue a list literal:

```
mylist = [1111, 2222, 3333]
```

- Parentheses are the catchall device—because any expression can be wrapped in them, simply inserting a left parenthesis allows you to drop down to the next line and continue your statement:

```
X = (A + B + C + D)
```

Statements

- You do not type anything explicit in your code to syntactically mark the beginning and end of a nested block of code.
  - You don’t need to include begin/end, then/endif, or braces around the nested block, as you do in C-like languages .

- Python uses the statements’ physical indentation to determine where the block starts and stops.

- The syntax rule is only that for a given single nested block, all of its statements must be indented the same distance to the right.
  - If this is not the case, you will get a syntax error
  - Python doesn’t care how you indent (you may use either spaces or tabs)
  - Python doesn’t care how much you indent (you may use any number of spaces or tabs).
  - There is no universal standard: four spaces or one tab per level is common

- It essentially means that you must line up your code vertically, in columns, according to its logical structure.

Statements

Assignment Statements

- In its basic form, you write the target of an assignment on the left of an equals sign, and the object to be assigned on the right.

- The target on the left may be a name or object component, and the object on the right can be an arbitrary expression that computes an object.

Statements

Assignment Statements

- Assignments create object references.
  - Assignments store references to objects in names or data structure components.
  - Python variables are more like pointers than data storage areas.
  - They always create references to objects instead of copying the objects.
- Names are created when first assigned.
  - Python creates variable name the first time you assign it a value
  - There’s no need to predeclare names ahead of time.
  - Once assigned, a name is replaced with the value it references whenever it appears in an expression.
- Names Must Be Assigned Before Being Referenced.
  - It’s an error use a name to which you haven’t yet assigned a value.
  - Python raises an exception if you try.

Statements

Operation	Interpretation
`spam = 'Spam'`	Basic form
`spam, ham = 'yum', 'YUM'`	Tuple assignment (positional)
`[spam, ham] = ['yum', 'YUM']`	List assignment (positional)
`a, b, c, d = 'spam'`	Sequence assignment, generalized
`a, *b = 'spam'`	Extended sequence unpacking (Python 3.X)
`spam = ham = 'lunch'`	Multiple-target assignment
`spams += 42`	Augmented assignment (equivalent to `spams = spams + 42`)

Statements

Expression Statements

In Python, you can use an expression as a statement, too. But because the result of the expression won't be saved, it usually makes sense to do so only if the expression does something useful as a side effect.

Expressions are commonly used as statements in two situations:

- For calls to functions and methods
  - Some functions and methods do their work without returning a value.
  - Such functions are sometimes called procedures in other languages.
  - Because they don't return values that you might be interested in retaining, you can call these functions with expression statements.

- For printing values at the interactive prompt

Statements

Operation	Interpretation
`spam(eggs, ham)`	Function calls
`spam.ham(eggs)`	Method calls
`spam`	Printing variables in the interactive interpreter
`print(a, b, c, sep='')`	Printing operations in Python 3.X
`yield x ** 2`	Yielding expression statements

Control Flow

The **if, while** and **for** statements implement traditional control flow constructs.

Control flow is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.

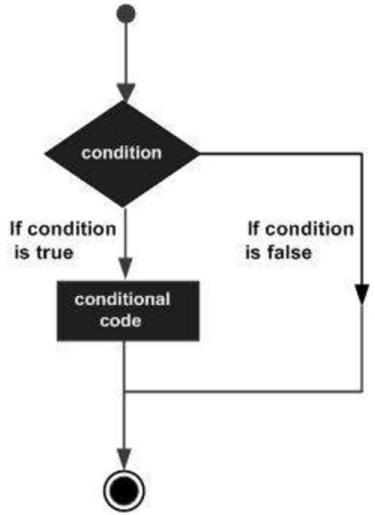
The emphasis on explicit control flow distinguishes an *imperative programming* language from a *declarative programming* language.

Within an imperative programming language, a *control flow statement* is a statement which execution results in a choice being made as to which of two or more paths to follow.

If Statement

The if statement is used for **conditional execution**.

It is used for selecting from alternative actions based on test results.



If Statement

General Format:

```
if test1 :
        Body

elif test2 :
        Body

 else :
        Body
```

- It selects exactly one of the bodies by evaluating the tests one by one until one is found to be true.
- That body is executed (and no other body is executed).
- If all expressions are false, the body of the else clause, if present, is executed.

If Statement

Ternary Expression:

```
if X:
    A = Y

else :
    A = Z
```

```
A = Y if X else Z
```

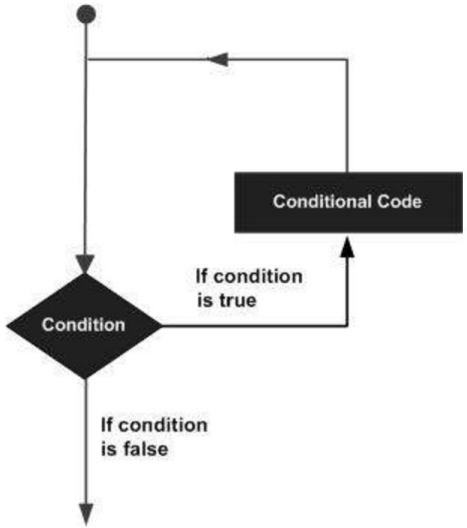
- The expression on the right has the same exact effect as the four-line if statement on the left.
- Python runs expression Y if X is true, and runs expression Z if X is false.

While Loop

Python’s most general iteration construct

The while statement is used for **repeated execution as long as a test is true**.

When the test becomes false, control passes to the statement that follows the while block.



While Loop

```
while test:
        Body

else :
        Body
```

This repeatedly tests the test expression and, if it is true, executes the body.

If the expression is false the suite of the else clause, if present, is executed and the loop terminates.

A break statement executed in the first body terminates the loop without executing the else clause’s body.

A continue statement executed in the first body skips the rest of the body and goes back to testing the expression.

Loop Control Statements

break

- Jumps out of the closest enclosing loop (past the entire loop statement)

continue

- Jumps to the top of the closest enclosing loop (to the loop's header line)

pass

- Does nothing at all: it's an empty statement placeholder

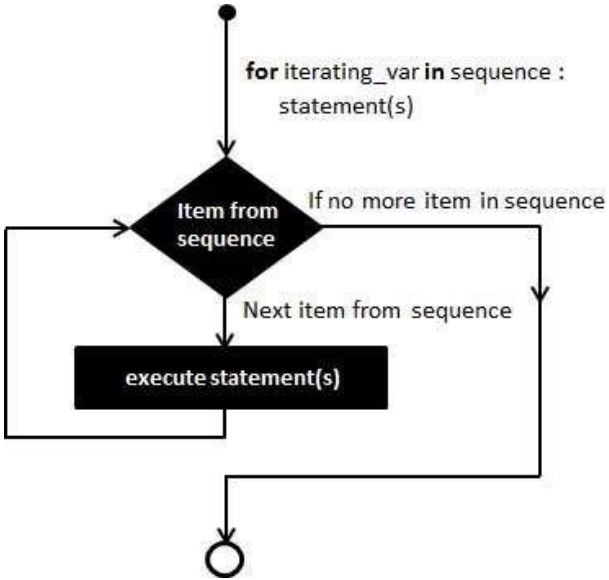
loop else block

- Runs if and only if the loop is exited normally (i.e., without hitting a break)

For Loop

The for loop is a generic iterator in Python.

It can step through items in any ordered sequence (string, tuple, list, etc.) or other iterable object



For Loop

```
for target in sequence/iterable object :
        Body
else :
        Body
```

The header line specifies an assignment target along with the object you want to step through.

When the for loop runs, it assigns the items in the iterable object to the target one by one and executes the loop body for each.

When the the end of the object is reached without encountering a break, the body in the else clause, if present, is executed, and the loop terminates.

A break statement executed terminates the loop without executing the else clause's body.

A continue statement executed in the first body skips the rest of the body and continues with the next item, or with the else clause if there is no next item.

Loop Coding Techniques

- The for loop is generally simpler to code and often quicker than a while loop.
- Resist the temptation to count things in Python - use iteration tools instead.
- Python provides a set of built-ins that allow you to specialize the iteration in a for loop.
  - Range function - produces a series of successively higher integers, which can be used as indexes in a for loop.
  - Zip function - returns a series of parallel-item tuples, which can be used to traverse multiple sequences in a for loop.
  - Enumerate function - generates both values and indexes of items in an iterable so we don't need to count manually.
  - Map function - calls a function on each item in an iterable
  - Filter function - returns items in an iterable for which a passed in function returns true

Iteration

- An object in Python is considered **iterable** if it is either:
  - Physically stored in sequence
  - It is an object that produces one result at a time in the context of an iteration tool like a for loop
- An object in Python is an **iterator** if it follows the **iteration protocol**:
  - The object has a **__next__** method to advance to the next result.
  - The method raises **StopIteration** at the end of the series of results.

Iteration

Full Iteration Protocol

- Two objects: iterable object, iterator object
    - You initialize iteration for an iterable object, **iter(obj) runs objects __iter__().**
    - This returns an iterator object.
    - The iterator produces the values during iteration, **next(obj) runs objects __next__()**.
    - The iterator raises **StopIteration** when it has stepped through the entire sequence.
- Some Python objects only support one iterator operating on it at a time, such as files. In this case the first step can be forgone because these object are their own iterators.
- Other Python objects support multiple iterators, such as lists. For these objects we need to create iterators using **iter**.

List Comprehension

- List comprehensions are written in square brackets(‘[]’) because they are ultimately a way to construct a new list.
  - They begin with an arbitrary expression.
  - Followed by a for loop header.

```
>>> L
[21, 22, 23, 24, 25]
```

Using Loops

```
>>> res = []
>>> for x in L:
...     res.append(x + 10)
...
>>> res
[31, 32, 33, 34, 35]
```

produces the same results

List Comprehension

```
L = [x + 10 for x in L]
```

- x iterates through L, and on each iteration the expression (x+10) is applied. The result is appended to a list.

List Comprehension

- List comprehension is never required. for loops can accomplish the same tasks.

However,

- List comprehensions are more Pythonic
- List comprehensions are optimized and can achieve up to double the speeds of equivalent for loops

List Comprehension

List comprehension can be extended and made more complex.

- We can create a filter clause using an if clause.
  - Per for clause we can have one if statement.

```
>>> lines = [line.rstrip() for line in open('script2.py') if line[0] == 'p']
>>> res = []
>>> for line in open('script2.py'):
...     if line[0] == 'p':
...         res.append(line.rstrip())
```

- We can nest more for clauses.

```
>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

>>> res = []
>>> for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
```

Programming

Algorithm

An algorithm is a step by step process that describes precisely how to solve a problem and/or complete a task, which will always give the correct result.

Pseudo-Code

Matches a programming language fairly closely, but leaves out details that could easily be added later by a programmer.

Pseudocode doesn't have strict rules about the sorts of commands you can use, but it's halfway between an informal instruction and a specific computer program.

Programming - Fizz Buzz

Write a program that prints the numbers from 1 to 100.

But for multiples of three print “Fizz” instead of the number

And for the multiples of five print “Buzz”.

For numbers which are multiples of both three and five print “FizzBuzz”."

Programming - Fizz Buzz Solution

```python
# Loop through integers: 1, 2, 3,...,100
for i in range(1, 101):

    # Test Divisibility of i by 3
    if (i % 3 == 0):
        print('Fizz', end = '')          # Suppress the newline w/ end = ''

    # Test Divisibility of i by 5
    if (i % 5 == 0):
        print('Buzz', end = '')

    # Test if i is NOT Divisible by both 3 and 5
    if (i % 3 != 0) and (i % 5 != 0):
        print(i, end = '')

    print()                              # Newline
```

```
slr:Desktop sammishra$ python3 fizzbuzz.py
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
```

This is not the only correct way to code a solution to this problem

Programming - Simple Primality Testing

The simplest primality test is *trial division*:

> Given an input number *n*, check whether any integer *m* from 2 to √n evenly divides *n* (the division leaves no remainder).
>
> If *n* is divisible by any *m* then *n* is composite, otherwise it is prime.

Task: Write a script that asks the user for a number and then uses trial division to test if that number is prime.

Hints: input(prompt) returns a string, before you do any math to this input you must convert this string to a numeric data type, do this with one line of code this way: int(input(prompt).

> pow(num, .5) will give you the square root of num
>
> use round(num) to round num to the nearest integer or int(num) to just drop the decimal.
>
> Keep in mind how range(start,stop,step) works and pay particular attention to how the stop value works.

Programming - Simple Primality Testing Solution

```
num = int(input('Enter A Number: '))
for i in range(2, num):
    if (num % i == 0):
        print('{} is composite'.format(num))
        quit()

print('{} is prime'.format(num))
```

Functions

Programming languages support decomposing problems in several different ways:

- Most programming languages are **procedural**:
  - programs are lists of instructions that tell the computer what to do with the program's input.
  - C, Pascal, and even Unix shells are procedural languages.

- In **declarative** languages:
  - You write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently.
  - SQL is the declarative language.

- **Functional** programming:
  - Decomposes a problem into a set of functions.
  - In a functional program, input flows through a set of functions.
  - Each function operates on its input and produces some output.

Functions

Python gives you many built-in functions but you can also create your own functions. These functions are called *user-defined functions.*

A function is a block of organized, reusable code. They group a set of statements so they can be run more than once in a program

Functions provide better modularity for your application and a high degree of code reusing.

Functions also provide a tool for splitting systems into pieces that have well-defined roles.

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Functions

def Statements

- The def statement creates a function object and assigns it a name.
  Its general form:

```
def name(arg1, arg2,... argN):
    statements
```

- Function header begins with the keyword **def** followed by the **function name** and **parentheses ( )**.
- Any **input parameters** or **arguments** should be placed within these parentheses.
- The code block within every function starts with a **colon :** and is **indented**.

Functions

def Statements

- **def** is an executable statement - it creates a new function object at runtime (as opposed to a compiled language)
- Like all other statements **def** statements can appear where statements are valid.
  - Nested function declarations are allowed

```
if test:
    def func():          # Define func this way
        ...
else:
    def func():          # Or else this way
        ...
...
func()                   # Call the version selected and built
```

```
othername = func         # Assign function object
othername()              # Call func again
```

Functions

```
>>> 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]
```

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

When Python reaches and runs this **def**, it creates a new function object that packages the function's code and assigns the object to the name **fib2** .

The function definition **DOES NOT** execute the function body.

To execute a function it must be **called**.

- You cannot call a function until after you have defined it.

You can **call** (run) the function in your program by adding parentheses after the function's name.

- The parentheses may optionally contain one or more object arguments

Functions

return leaves the current function call with the expression list.

Even functions without a **return** statement do return a value, `None`.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring; it's good practice to include docstrings in code that you write, so make a habit of 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]
```

```
>>> 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
```

Functions

argument

A value passed to a function when calling the function. There are two kinds of argument:

- ***keyword argument***: an argument preceded by an **identifier** (e.g. `name=`) in a function call or be passed as values in a **dictionary preceded by** `**`.
  - For example, `3` and `5` are both keyword arguments in the following calls to complex():

    `complex(real=3, imag=5)`

- ***positional argument***: an argument that is not a keyword argument. Positional arguments can appear at the **beginning** of an argument list or be passed as elements of an **iterable preceded by** `*`.
  - For example, `3` and `5` are both positional arguments in the following calls:

    `complex(3, 5)`

Functions

Default Argument Values

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
```

Keyword Arguments

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
```

Arbitrary Argument Lists

```
def write_multiple_items(file, separator, *args):
```

Functions

Default Argument Values

When one or more parameters have the form *parameter* `=` *expression*, the function is said to have “default parameter values.”

For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter’s default value is substituted.

If a parameter has a default value, all following parameters must also have a default value.

Functions

Default Argument Values

```
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)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Functions

Default Argument Values

Default parameter values are evaluated from left to right when the function definition is executed.

The default value is evaluated only once.

This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

```
[1]
[1, 2]
[1, 2, 3]
```

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

```
slr:Desktop sammishra$ python3 function.py
[1]
[2]
[3]
slr:Desktop sammishra$
```

Functions

Keyword Arguments

The following function accepts one required argument (voltage) and three optional arguments (state, action, and type).:

```
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, "!")
```

This function can be called in any of the following ways:

```
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
```

Functions

Keyword Arguments

The following function accepts one required argument (voltage) and three optional arguments (state, action, and type).:

```
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, "!")
```

All the following calls would be invalid:

```
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
```

Functions

Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments.

- Formal parameter of the form `**name` - receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter.
- Formal parameter of the form `*name` - receives a tuple containing the positional arguments beyond the formal parameter list.

```
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
```

Palindromes

A **palindrome** is a word which reads the same backward as forward, such as *madam* or *racecar*.

Task: Given a word as input determine if it is a palindrome. Return True if the word is a palindrome, False if it is not.

Palindromes

```
def palindrome(forward):

    print(forward)
    backward = ''.join(reversed(forward))
    print(backward)

    if backward == forward:
        return True
    else:
        return False

def main():
    word = 'mommy'
    print(palindrome(word))

main()
```

Palindromes

```
def palindrome(forward):

    print(forward)
    backward = forward
    backward = list(backward)
    backward.reverse()
    backward = ''.join(backward)
    print(backward)

    if backward == forward:
        return True
    else:
        return False

def main():
    word = 'mommy'
    print(palindrome(word))

main()
```

Palindromes

palindromes.py

```
def get_word():
    '''Get the word to be checked from user input'''

    word = input("Enter A Word: ")
    return word

def check_word(word):
    '''Check if the word is a palindrome'''

    length = len(word)
    for i in range(length//2):
        left = i
        right = -(i+1)
        if word[left] != word[right]:
            return False
        return True

def main():
    print("This script checks if a word is a Palindrome.")
    word = get_word()
    print(check_word(word))

main()
```

Anagrams

An **anagram** is the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once; for example, the word anagram can be rearranged into "nag a ram".

Task: Given two words as input determine if they are an anagram of one-another. Return True if the word is an anagram, False if it is not.

Anagrams

```
def check_words(word1, word2):
    '''Check if two words are anagrams of one-another'''

    if len(word1) != len(word2):
        return False

    dict1 = {}
    dict2 = {}

    for i in word1:
        if i not in dict1:
            dict1[i] = 0
        else:
            dict1[i] += 1

    for i in word2:
        if i not in dict2:
            dict2[i] = 0
        else:
            dict2[i] += 1

    if dict1 == dict2:
        return True
    else:
        return False
```

```
def get_words():
    '''Get the words to be checked from user input'''

    word1 = input("Enter A Word: ")
    word2 = input("Enter A Word: ")

    return word1, word2
```

```
def main():
    print('This script checks if two words are anagrams of each other.')
    word1, word2 = get_words()
    print(check_words(word1, word2))

main()
```

Variable Scope

- A variable which is defined in the main body of a file is called a **global variable**.
  - It will be visible throughout the file, and also inside any file which imports that file.

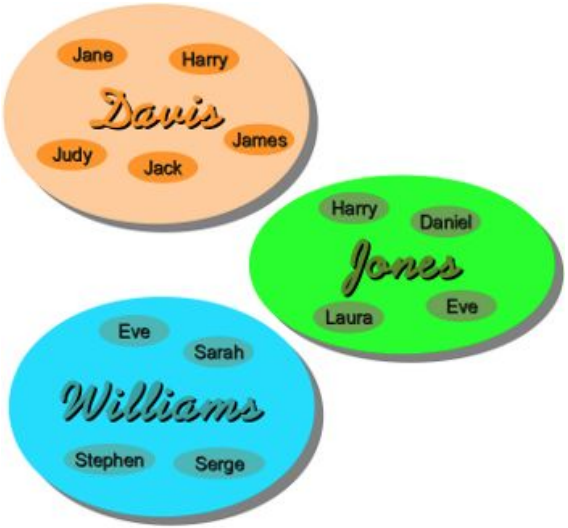
- A variable which is defined inside a function is **local to that function**.
  - When we use the assignment operator (`=`) inside a function, it creates a new local variable.
  - It is accessible from the point at which it is defined until the end of the function.
  - The parameter names in the function definition behave like local variables.

Variable Scope

```
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1

def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

Variable Scope

- *A* **namespace** is a mapping from names to objects.
- Generally speaking, a namespace is a naming system for making names unique to avoid ambiguity.
  - A namespacing system from daily life - the naming of people in firstname and surname.



Variable Scope

- Many programming languages use namespaces for **identifiers**.
  - An identifier defined in a namespace is associated with that namespace. This way, the same identifier can be independently defined in multiple namespaces.

- Examples of namespaces:
  - the set of built-in names
  - the global names in a module
  - the local names in a function invocation
  - In a sense the set of attributes of an object also form a namespace

- The important thing to know about namespaces is that there is **absolutely no relation between names in different namespaces**
  - for instance, two different modules may both define a function maximize without confusion — users modules must prefix it with the module name.

Variable Scope

Lifetime of a Namespace

- Not every namespace is accessible (or alive) at any moment during the execution of the script.
- There is one namespace which is present from beginning to end: The namespace containing the built-in names.
- The global namespace of a module is generated when the module is read in.
- Module namespaces normally last until the script ends.
- When a function is called, a local namespace is created for this function. This namespace is deleted either if the function ends, i.e. returns, or if the function raises an exception, which is not dealt with within the function.

Variable Scope

Scopes

The scope of a namespace is the area of a program where this name can be unambiguously used, for example inside of a function.

At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the **local names**
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains **non-local, but also non-global names**
- the next-to-last scope contains the current module's **global names**
- the outermost scope (searched last) is the namespace containing **built-in names**

Variable Scope

- If a name is declared **global**, then all references and assignments go directly to the middle scope containing the module's global names.

- To rebind variables found outside of the innermost scope, the **nonlocal** statement can be used
  - if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

- Usually, the local scope references the local names of the current function.
  - Outside functions, the local scope references the same namespace as the global scope: the module's namespace.
  - Class definitions place yet another namespace in the local scope.

Variable Scope

This is an example demonstrating how to reference the different scopes and namespaces, and how global and nonlocal effect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Variable Scope

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Note how the *local* assignment (which is default) didn't change *scope_test*'s binding of *spam*.
The nonlocal assignment changed *scope_test*'s binding of *spam*
The global assignment changed the module-level binding.
You can also see that there was no previous binding for *spam* before the global assignment.

Functions

Recursion

Functions

First Class Functions

Functions

Lambda

Functions

Generator Functions

File I/O

Before you can read or write a file, you have to open it using Python's built-in *open()* function.

This function creates a **file** object, which would be utilized to call other support methods associated with it.
It is most commonly used with two arguments: `open(filename, mode)`.

> *filename* is a path-like object giving the pathname (absolute or relative to the current working directory) of the file to be opened.
>
> *mode* is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode.

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open a disk file for updating (reading and writing)
`'U'`	universal newlines mode (deprecated)

File I/O

- To read a file's contents, call *fileobject*.read(size), which reads some quantity of data and returns it as a string .
  - *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned.

- *fileobject*.readline() reads a single line from the file.

- For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in fileobject:
...         print(line)
...
This is the first line of the file.
Second line of the file
```

File I/O

- *fileobject*.seek(offset, reference point) moves to the position that is computed from adding *offset* to a *reference point*.
  - A *reference point* of 0 measures from the beginning of the file
  - 1 uses the current file position
  - 2 uses the end of the file as the reference point
  - r*eference point* can be omitted and defaults to 0.

- *fileobject*.write(**string**) writes the contents of ***string*** to the file, returning the number of characters written.

File I/O

- When you want to save more complex data types like nested lists and dictionaries Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation). The standard module called json can take Python data hierarchies, and convert them to string representations.
  - This process is called *serializing*.
  - Reconstructing the data from the string representation is called *deserializing*.

- If you have an object x, you can view its JSON string representation with a simple line of code:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

- Another variant of the dumps() function, called dump(), simply serializes the object to a text file.
So if f is a text file object opened for writing, we can do this:

```
json.dump(x, f)
```

- To decode the object again, if f is a text file object which has been opened for reading:

```
x = json.load(f)
```

- This simple serialization technique can handle lists and dictionaries.

File I/O

- Call `f.close()` to close the file and free up any system resources used by it.
  - If you don't explicitly close a file, Python's garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while.

- It is good practice to use the with keyword when dealing with file objects.
  - The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.

```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

Fibonacci

the **Fibonacci numbers** are the numbers characterized by the fact that every number after the first two is the sum of the two preceding ones:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

Task: Given an integer as input, determine the fibonacci sequence of that integer. Use recursion (see following slides).

i.e.	fib(1)	returns 0
	fib(5)	returns 0,1,1,2,3
	fib(13)	returns 0,1,1,2,3,5,8,13,34,55,89,144

Fibonacci

Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body.

Termination condition:
A recursive function has to terminate to be used in a program. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case.
A base case is a case, where the problem can be solved without further recursion.
A recursion can lead to an infinite loop, if the base case is not met in the calls.

Example:
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1

Replacing the calculated values gives us the following expression
4! = 4 * 3 * 2 * 1

Fibonacci

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial(" ,n-1, "): ",res)
        return res

print(factorial(5))
```

Output

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for  2  * factorial( 1 ):  2
intermediate result for  3  * factorial( 2 ):  6
intermediate result for  4  * factorial( 3 ):  24
intermediate result for  5  * factorial( 4 ):  120
120
```

Fibonacci

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fib_seq(n):
    for i in range(n):
        if i == n-1:
            print(fib(i))
        else:
            print(fib(i), end =', ')

def main():
    n = int(input('Enter A Number: '))
    fib_seq(n)

main()
```

Python Conceptual Hierarchy

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

Import Statement

- Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter.
- Such a file is called a ***module***; definitions from a module can be ***imported*** into other modules.
- A module can contain executable statements as well as function definitions.
  - These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement.(They are also run if the file is executed as a script.)
- It is customary but not required to place all import statements at the beginning of a module (or script, for that matter).

Import Statement

“fibo.py”

```
# Fibonacci numbers module

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

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

Using the module name you can now access the functions:

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

Import Statement

“fibo.py”

```
# Fibonacci numbers module

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

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

Here is a variant of the import statement that imports names from a module directly. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken (so in the example, fibo is not defined).

Import Statement

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_). In most cases Python programmers do not use this facility.

Note that in general the practice of importing * from a module or package is frowned upon.

Import Statement

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`.
That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

Import Statement

The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```

Without arguments, dir() lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Built-In Libraries - datetime

A datetime object is a single object containing all the information from a date object and a time object.

Constructor:

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

The year, month and day arguments are required.

classmethod datetime.**now**(*tz=None*)

Return the current local date and time.

Instance methods:

datetime.**date**()

Return date object with same year, month and day.

datetime.**time**()

Return time object with same hour, minute, second, microsecond and fold.

datetime.**strftime**(*format*)

Return a string representing the date and time

Built-In Libraries - datetime

Directive	Meaning	Example	Notes
`%a`	Weekday as locale's abbreviated name.	Sun, Mon, ..., Sat (en_US);	(1)
`%A`	Weekday as locale's full name.	Sunday, Monday, ..., Saturday (en_US);	(1)
`%w`	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, ..., 6	
`%d`	Day of the month as a zero-padded decimal number.	01, 02, ..., 31	
`%b`	Month as locale's abbreviated name.	Jan, Feb, ..., Dec (en_US);	(1)
`%B`	Month as locale's full name.	January, February, ..., December (en_US);	(1)
`%m`	Month as a zero-padded decimal number.	01, 02, ..., 12	
`%y`	Year without century as a zero-padded decimal number.	00, 01, ..., 99	
`%Y`	Year with century as a decimal number.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
`%H`	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23	
`%I`	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12	
`%p`	Locale's equivalent of either AM or PM.	AM, PM (en_US);	(1), (3)
`%M`	Minute as a zero-padded decimal number.	00, 01, ..., 59	
`%S`	Second as a zero-padded decimal number.	00, 01, ..., 59	(4)
`%Z`	Time zone name (empty string if the object is naive).	(empty), UTC, EST, CST	
`%c`	Locale's appropriate date and time representation.	Tue Aug 16 21:30:00 1988 (en_US);	(1)
`%x`	Locale's appropriate date representation.	08/16/88 (None); 08/16/1988 (en_US);	(1)
`%X`	Locale's appropriate time representation.	21:30:00 (en_US);	(1)
`%%`	A literal `'%'` character.	%	

Built-In Libraries - datetime

```
>>> dt = datetime.datetime(2006, 11, 21, 16, 30)

>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'

 >>> dt.strftime("%x %X")

>>> datetime.datetime.now()
```

Built-In Libraries - random

random.**randrange**(*start*, *stop*[, *step*])
Return a randomly selected element from `range(start, stop, step)`.

random.**choice**(*seq*)
Return a random element from the non-empty sequence *seq*.

random.**choices**(*population*, *weights=None*, *k=1*)
Return a *k* sized list of elements chosen from the *population* with replacement.
If a *weights* sequence is specified, selections are made according to the relative weights.

random.**shuffle**(*x*)
Shuffle the sequence *x* in place.
To shuffle an immutable sequence and return a new shuffled list, use `sample(x, k=len(x))` instead.

random.**sample**(*population*, *k*)
Return a *k* length list of unique elements chosen from the population sequence or set. Used for random sampling without replacement.

Built-In Libraries - random

```
>>> randrange(10)                       # Integer from 0 to 9 inclusive
7

>>> choice(['win', 'lose', 'draw'])     # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                       # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)   # Four samples without replacement
[40, 10, 50, 30]
```

Built-In Libraries - statistics

`mean()`	Arithmetic mean (“average”) of data.
`median()`	Median (middle value) of data.
`mode()`	Mode (most common value) of discrete data.

`stdev()`	Sample standard deviation of data.
`variance()`	Sample variance of data.

Built-In Libraries - csv

The CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases.

csv.**reader**(*csvfile*)
Return a reader object which will iterate over lines in the given *csvfile*.

csv.**writer**(*csvfile*)
Return a writer object responsible for converting the user's data into delimited strings on the given file-like object.

class csv.**DictReader**(*f, fieldnames*)
Create an object that operates like a regular reader but maps the information in each row to an OrderedDict whose keys are given by the optional *fieldnames* parameter.

class csv.**DictWriter**(*f, fieldnames*)
Create an object which operates like a regular writer but maps dictionaries onto output rows.The *fieldnames* parameter is a sequence of keys that identify the order in which values in the dictionary passed to the writerow() method are written to file *f*.

Built-In Libraries - csv

```
import csv

with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ', quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])

with open('eggs.csv', newline='') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
        for row in spamreader:
            print(', '.join(row))
```

Built-In Libraries - csv

```
import csv

with open('names.csv', 'w') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Eric', 'last_name': 'Idle'})
    writer.writerow({'first_name': 'John', 'last_name': 'Cleese'})
```

```
>>> with open('names.csv') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
```

Object Oriented Programming

Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data (functional programming). The programming challenge was seen as how to write the logic, not how to define the data.

Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them.

Examples of objects range from human beings (described by name, address, and so forth) to buildings and floors (whose properties can be described and managed) down to the little widgets on a computer desktop (such as buttons and scrollbars).

OOP is a programming paradigm based on the concept of "**objects**", which may contain:

- data, often known as **attributes**
- and code, often known as **method**

Object Oriented Programming

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Method :** A special kind of function that is defined in a class definition.
- **Instantiation:** The creation of an instance of a class.
- **Instance:** An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.
- **Object:** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Object Oriented Programming - Class Definition

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class definitions, like function definitions must be executed before they have any effect.

In practice, the statements inside a class definition will usually be function definitions.

Object Oriented Programming - Attribute References

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`.

So, if the class definition looked like this:

```
class MyClass:
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively.

Class attributes can also be assigned to, so you can change the value of `MyClass.i`

Object Oriented Programming - Class Instantiation

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

Object Oriented Programming - Class Instantiation

Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named __init__(), like this:

```
def __init__(self):
    self.data = []
```

When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the __init__() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to __init__(). For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Object Oriented Programming - Instance Objects

The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes need not be declared; like local variables, they spring into existence when they are first assigned to.

For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

Object Oriented Programming - Instance Objects

The other kind of instance attribute reference is a *method*.

A method is a function that “belongs to” an object.

Usually, a method is called right after it is bound:

```
x.f()
```

In the MyClass example, this will return the string 'hello world'.

However, it is not necessary to call a method right away: x.f is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

will continue to print hello world until the end of time.

Object Oriented Programming - Instance Objects

You may have noticed that `x.f()` was called without an argument above, even though the function definition for f() specified an argument. What happened to the argument? .

The special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`.

Object Oriented Programming

```
class Dog:

    kind = 'canine'                 # class variable shared by all instances

    def __init__(self, name):
        self.name = name            # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                          # shared by all dogs
'canine'
>>> e.kind                          # shared by all dogs
'canine'
>>> d.name                          # unique to d
'Fido'
>>> e.name                          # unique to e
```

Object Oriented Programming

```
class Dog:

    tricks = []                 # mistaken use of a class variable

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

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                    # unexpectedly shared by all dogs
['roll over', 'play dead']
```

The *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances.

Object Oriented Programming

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)
```

```
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

Object Oriented Programming - Inheritance

The syntax for inheritance definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    <statement-N>
```

When the class object is constructed, the base class is remembered.
This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class.
There's nothing special about instantiation of derived classes: DerivedClassName() creates a new instance of the class.

Derived classes may override methods of their base classes.
There is a simple way to call the base class method directly: just call BaseClassName.methodname(self, arguments).

Python has two built-in functions that work with inheritance:

- Use isinstance() to check an instance's type: isinstance(obj, int) will be True only if obj.__class__ is int or some class derived from int.
- Use issubclass() to check class inheritance: issubclass(bool, int) is True since bool is a subclass of int. However, issubclass(float,int) is False since float is not a subclass of int.

Object Oriented Programming

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr
```

```
class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

Object Oriented Programming

```
class Parent:        # define parent class
   def myMethod(self):
       print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
       print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```