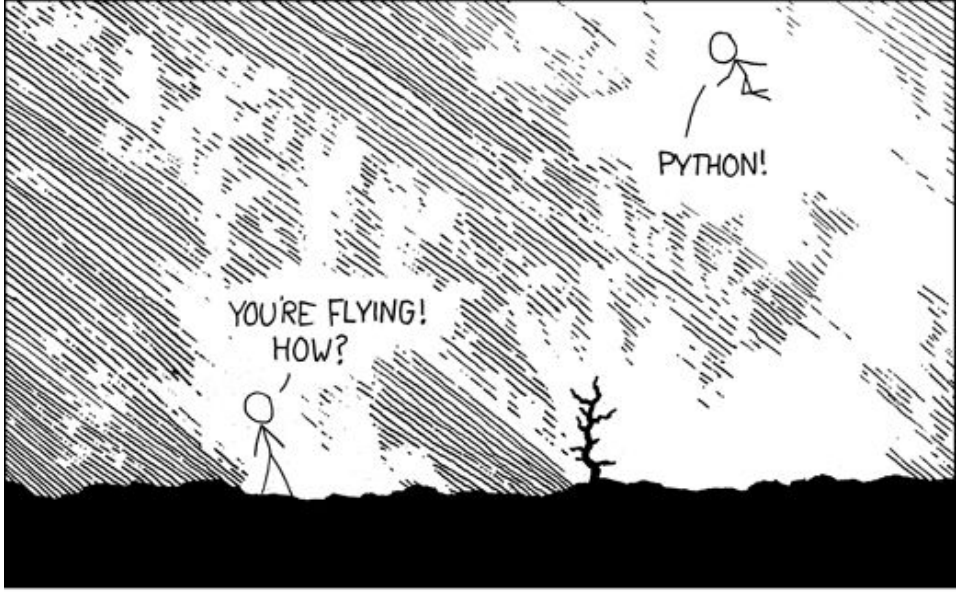
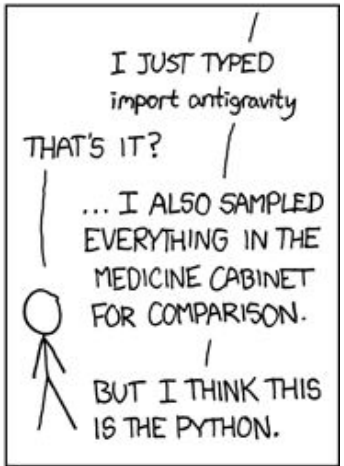

PYTHON!
YOU'RE FLYING! HOW?


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

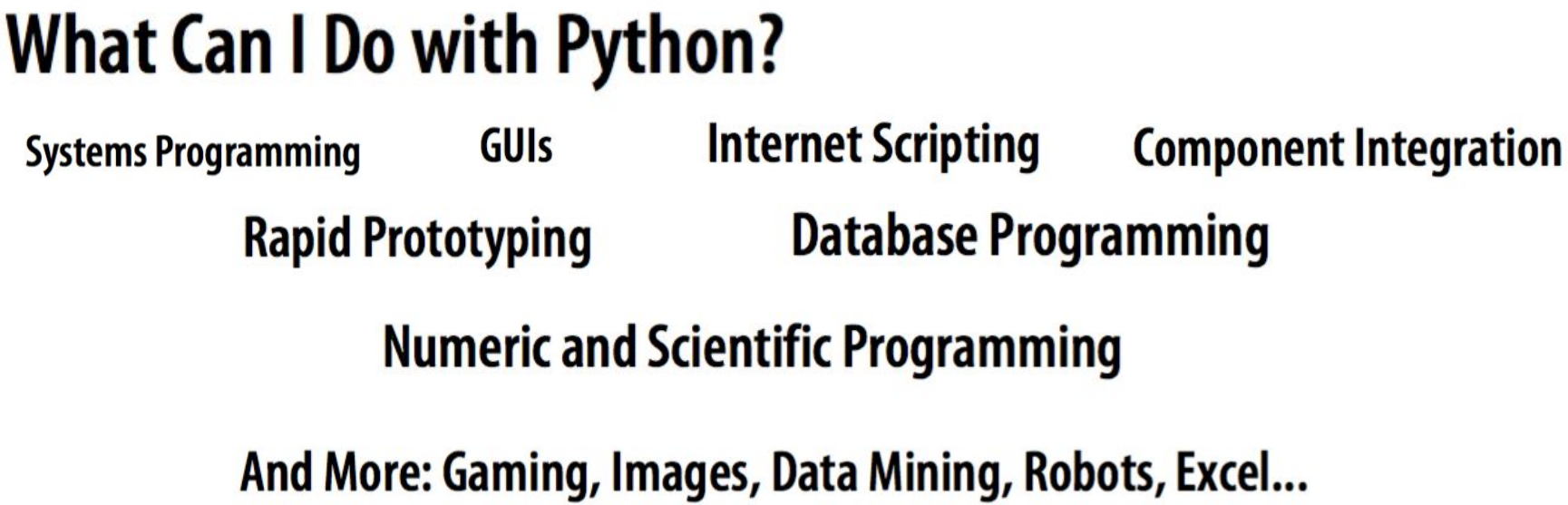
> Depending on which shell you use you'll have to edit either ~/.profile, ~/.zshrc, or ~/.bash_profile using a command line text editor such as vim
> Add this line:

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

> Save your changes

Python

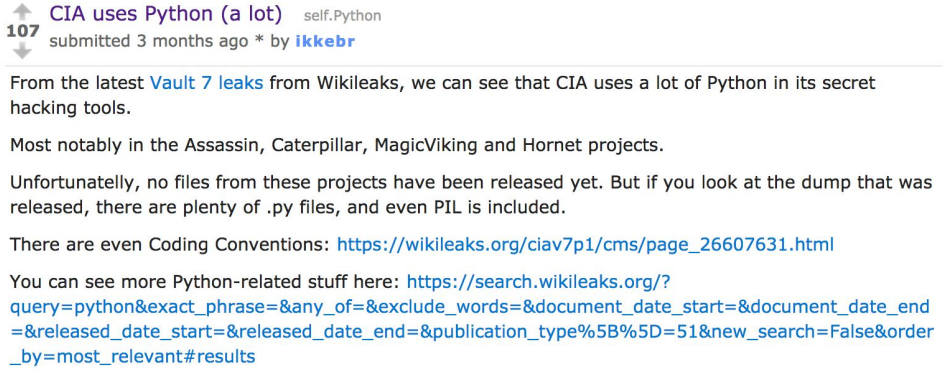
Python is a **high-level programming language** for **general-purpose programming**.
An **interpreted language**, Python has a design philosophy which emphasizes **code readability**.



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

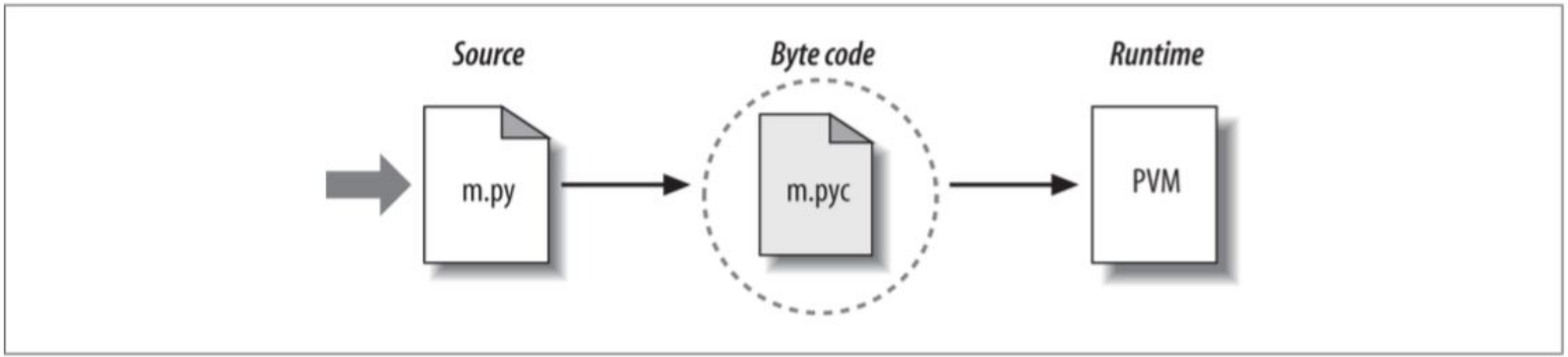
Benefits of Python:

- *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

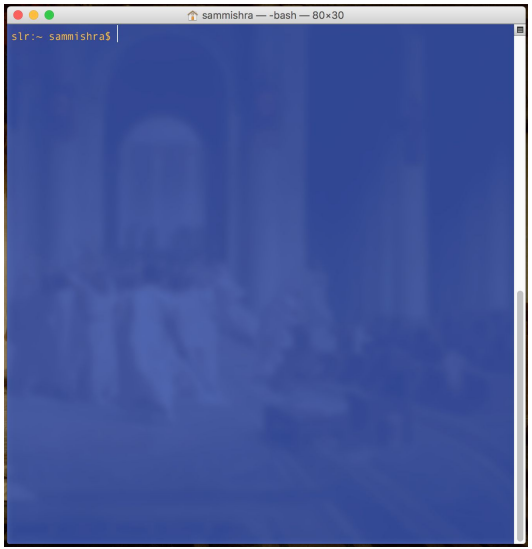
The Zen of Python

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

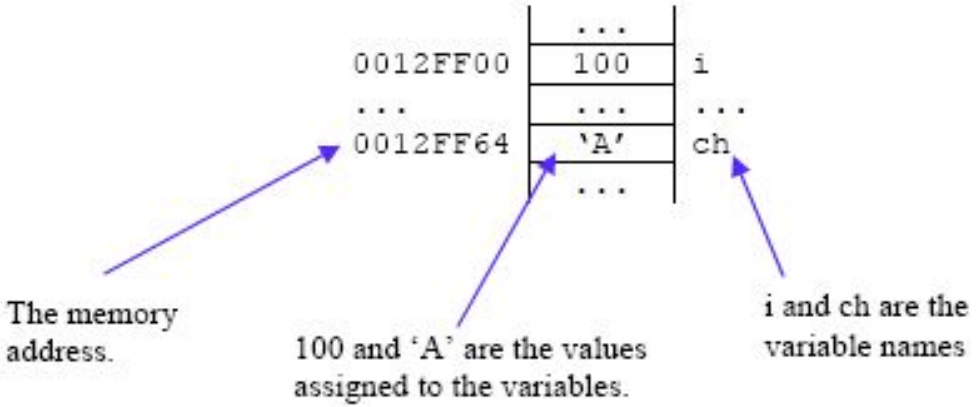
Python Conceptual Hierarchy

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:

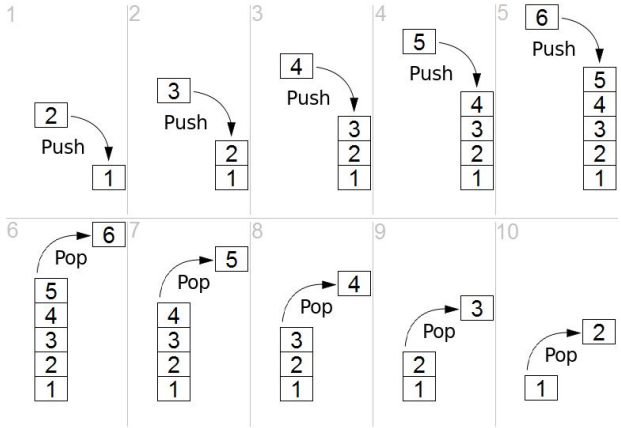
```
>>> L = ['obj1', 'obj2', 'obj3']
>>> L.append('obj4')
>>> L
['obj1', 'obj2', 'obj3', 'obj4']
```

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']
```

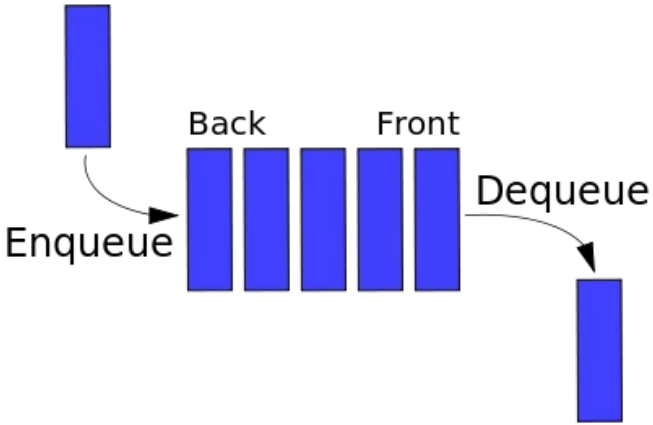
```
>>> L.pop()
'sally'
>>> L
['alice', 'bob']
>>> L.pop()
'bob'
>>> L
['alice']
>>> L.pop()
'alice'
>>> L
[]
```

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.

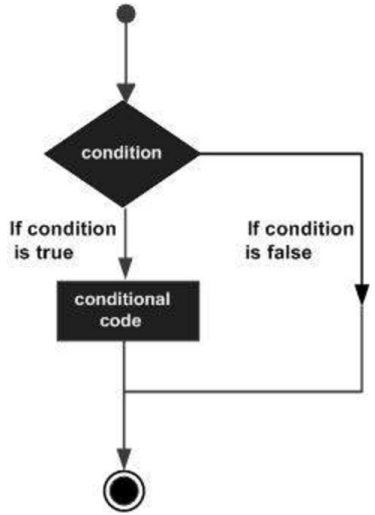
The emphasis on explicit control flow distinguishes an *imperative programming* language from a *declarative programming* language.

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
```

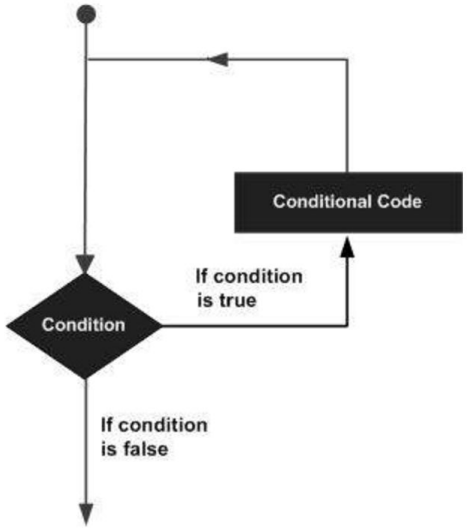
- The expression on the right has the same exact effect as the four-line if statement on the left.
- Python runs expression Y if X is true, and runs expression Z if X is false.

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

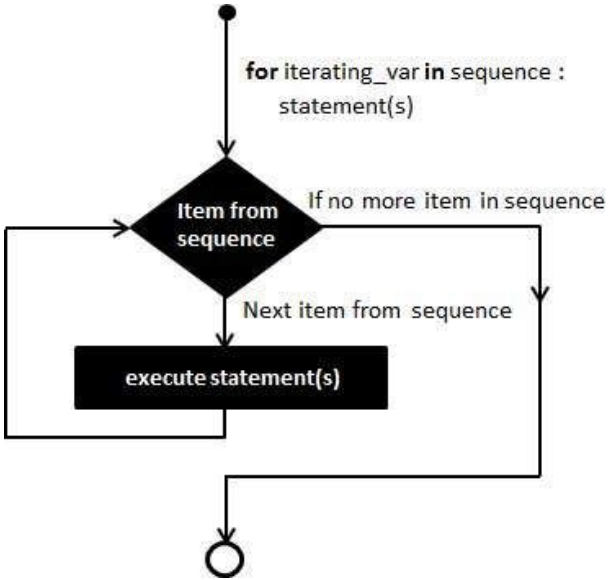
loop else block

- 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.