

Everything is an Object

- **Each object x has:**
 - Unique identifier, an integer, returned by `id(x)` ; think memory -> FIXED
 - Type, returned by `type(x)` -> FIXED
 - Some content (value) -> may or may not be changed (“mutated”) [a value can itself consist of one or more objects!]
 - Zero or more methods provided by type object to get and/or change content
 - Zero or more names (aliases)
- **Names are NOT properties of an object:**
 - They are stored in separate namespaces
 - Dictionaries of name \rightarrow object reference (id) pairs

Types are NOT properties of a name

Python Identifiers (Names)

Names used to identify things in Python:

- Variable, function, class, module, or other object

Identifier in Python 2:

- Starts with a letter A to Z or a to z or an underscore (`_`)
- Followed by zero or more letters, underscores, and digits (0 to 9)
- No punctuation characters such as `@`, `$` or `%` are allowed
- No limit on length
- Case-sensitive

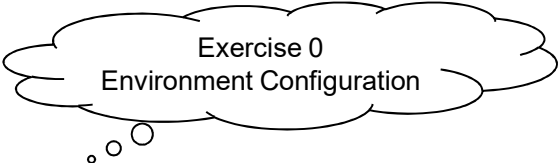
Python 3 allows Unicode codepoints from outside ASCII range:

- So you could rename the summation function: Σ = sum
- Not all codepoints: e.g. not $\sqrt{}$

Python Reserved Words

- **May not be used as constant or variable or other identifier name**

| | | |
|----------|---------|--------|
| and | exec | not |
| assert | finally | or |
| break | for | pass |
| class | from | print |
| continue | global | raise |
| def | if | return |
| del | import | try |
| elif | in | while |
| else | is | with |
| except | lambda | yield |



Exercise 0
Environment Configuration

Naming conventions

- `_*`** Names starting with underscore are treated as “private” and not imported by `from module import *`.
- `__`** **`__`** If starts and ends with a double underscore: name of “special method”. Defined by interpreter and its implementation (including standard library).
- `__`** ***** Class-private names: when used within the context of a class definition, re-written to mangled form to help avoid name clashes between “private” attributes of base and derived classes.

Stylistic (PEP8) conventions:

- Use underscores to separate multi-word names: `name_of_some_variable`
- Capitalize classnames: `Some_Class`

Comments in Python

- **Comments starts with:**

- A hash sign (#) that is not inside a string literal
- All characters after the # and up to the physical line end are part of the comment

```
# First comment  
print("Hello, Python!") # second comment
```

- **Will produce following result:**

```
Hello, Python!
```

- **Comment may be on same line after statement or expression:**

```
name = "Einstein" # This is again comment
```

- **Comment multiple lines as follows:**

```
# This is a comment.  
# This is a comment, too.  
# This is a comment, too.  
# I said that already.
```

Lines and Indentation

- **Blocks of code are denoted by line indentation:**
 - No braces for blocks of code for class and function definitions or flow control

- **Number of spaces in the indentation is variable:**
 - All statements within the block must be indented the same amount

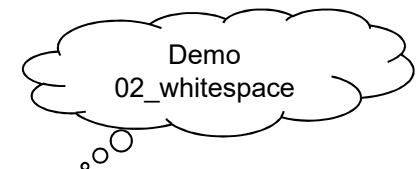
```
if True:                # Both blocks in first example are fine
    print("True")
else:
    print("False")
```

- **Other example:**

```
if True:                # Second line in second block will generate an error
    print("Answer")
    print("True")
else:
    print("Answer")
    print("False")
```

- `print()` **without new line is written as follows:**

```
print('hello python', end='')
```



Multi Line Statements

- **Statements in Python typically end with a newline**
- **Python allows the use of the line continuation character (\):**

- Denotes that the line should continue

```
total = item_one + \  
        item_two + \  
        item_three
```

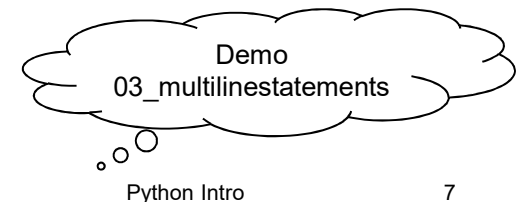
- **Statements contained within the [], {}, or () brackets:**

- No need to use the line continuation character

```
days = ['Monday', 'Tuesday', 'Wednesday',  
        'Thursday', 'Friday']
```

- **On a single line:**

- Semicolon (;) allows multiple statements on single line
- Neither statement should start a new code block



Python Data Types

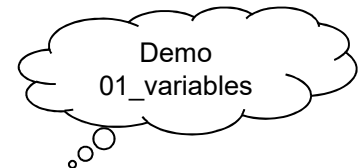
- **Data stored in memory can be of many types:**
 - Persons age is stored as a numeric value
 - Persons address is stored as alphanumeric characters
- **A type (or class) determines:**
 - the operations possible on them
 - storage method for each of them
- **Important Python data types:**
 - Numbers
 - String
 - List
 - Tuple
 - Set
 - Dictionary



Variables

- **Variables (names) are introduced using some form of assignment:**
 - They always refer to some object
 - Names can be erased (`del name`)
but object will persist as long as it “has” other names (aliases)
- **Equal sign = is used to assign values to variables:**
 - Operand to the left of the = operator is the name of the variable
 - Operand to the right of the = operator is the value stored in the variable

```
counter = 100          # An integer assignment
miles    = 1000.0       # A floating point
name     = "John"       # A string
```



Multiple Assignment

- **Simultaneous assignment of values:**

```
a = b = c = 1
```

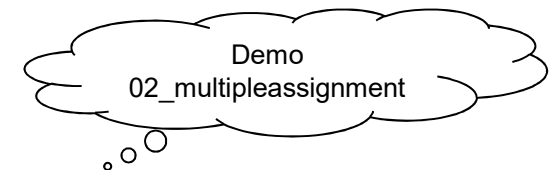
- **Integer object is created with the value 1:**

- All three variables are assigned to (refer to) the same memory location

- **Assignment of multiple objects to multiple variables:**

```
a, b, c = 1, 2, "john"
```

- Integer object with value 1 assigned to variable a
- Integer object with value 2 assigned to variable b
- String object with value "john" assigned to variable c



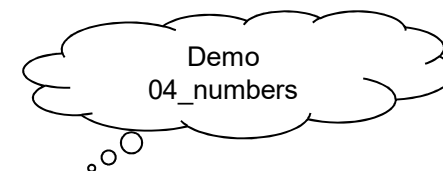
Numerical Types

- **Python supports different types to store numeric values:**
- `int` (signed integers):
 - Integers or ints, are positive or negative whole numbers with no decimal point
 - They are of unlimited size, can also be represented in octal and hexadecimal
- `float` (floating point real values):
 - Floats, represent real numbers
 - Written with a decimal point dividing the integer and fractional parts
- `complex` (complex numbers):
 - Have form $a + bJ$, with a and b floats and J (or j) represents the square root of i
 - a is the real part of the number and b is the imaginary part
 - Complex numbers are not used much in Python programming
- **Boolean type is a subtype of the integer type:**
 - Boolean values behave like values 0 and 1
 - When converted to a string in string context `"False"` or `"True"` are returned

Number Examples

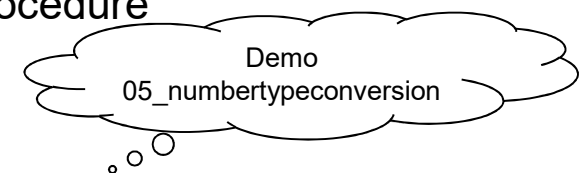
| int | float | complex |
|---------------------------|------------|------------|
| 10 | 0.0 | 3.14j |
| -100 | 15.20 | 45.j |
| 535633629843 | -21.9 | 9.322e-36j |
| 11 = 0b01011 = 0o13 = 0xB | 32.3e18 | .876j |
| 0xDEFABCECBDAE | -90. | -.6545+0j |
| 0x69 | -32.54e100 | 3e+26j |
| -0x260 | 70.2E-12 | 4.53e-7j |

- **Python 2 had separate `int` and `long` types for integer numbers:**
 - In Python 3, there is only one `int` type, which behaves like `long` type in Python 2
- **All integers except 0, have meaning `True`:**
 - Only 0 behaves as `False`



Type conversion: constructing new objects

- **Built-in constructors to perform conversion between data types:**
 - Functions return a new object representing the converted value
- `int(x [,base]) :`
 - Converts x to a plain integer, base specifies the base if x is a string
- `complex(x) :`
 - Converts x to a complex number with real part x and imaginary part zero
- `complex(x, y) :`
 - Converts x and y to a complex number with real part x and imaginary part y
 - x and y are numeric expressions
- `float(x) :`
 - Converts x to a floating-point number
- `bool([x]) :`
 - Convert value to Boolean using standard truth testing procedure



Strings in Python

- **Python uses quotes to denote string literals:**

- single ('), double (") and triple (""" or """)
- Same type of quote should start and end the string
- Triple quotes can be used to span the string across multiple lines

```
word = 'word'
```

```
sentence = "This is a sentence."
```

```
paragraph = """This is a paragraph.
```

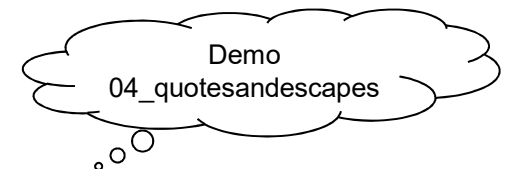
```
    It is made up of multiple sentences."""
```

```
paragraph = "This is a paragraph. " "It is made up of multiple sentences."
```

- **Single quote in a single quoted string:**

- Displayed with an escape character \

```
'What \'s your name'
```



Python Strings

- **no character type:**
 - Characters are treated as strings of length one, thus also considered a substring
- **immutable data types like numbers:**
 - Changing the value of a string data type results in a newly allocated object
- **strings are sequences, sharing sequence operators with e.g. lists:**
 - slice operator is used to retrieve subsets from sequence:
 - to get the third character: `"abcde"[2]` will return `"c"`
 - to test for membership: `'d' in "abcde"` will return `True`

String Formatting

Operator `%` when used with strings is the string format operator:

```
print("My name is %s and weight is %d kg!" % ('Albert', 55))
```

Result:

```
My name is Albert and weight is 55 kg!
```

In Python 3 new style of formatting available with `format()`:

- Allows complex variable substitutions and value formatting

```
s = '{0}, {1}, {2}'.format('a', 'b', 'c')
```

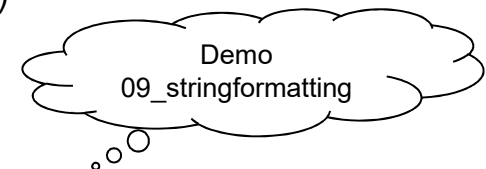
```
print(s) => 'a, b, c'
```

```
s = '{}', {}, {}'.format('a', 'b', 'c')
```

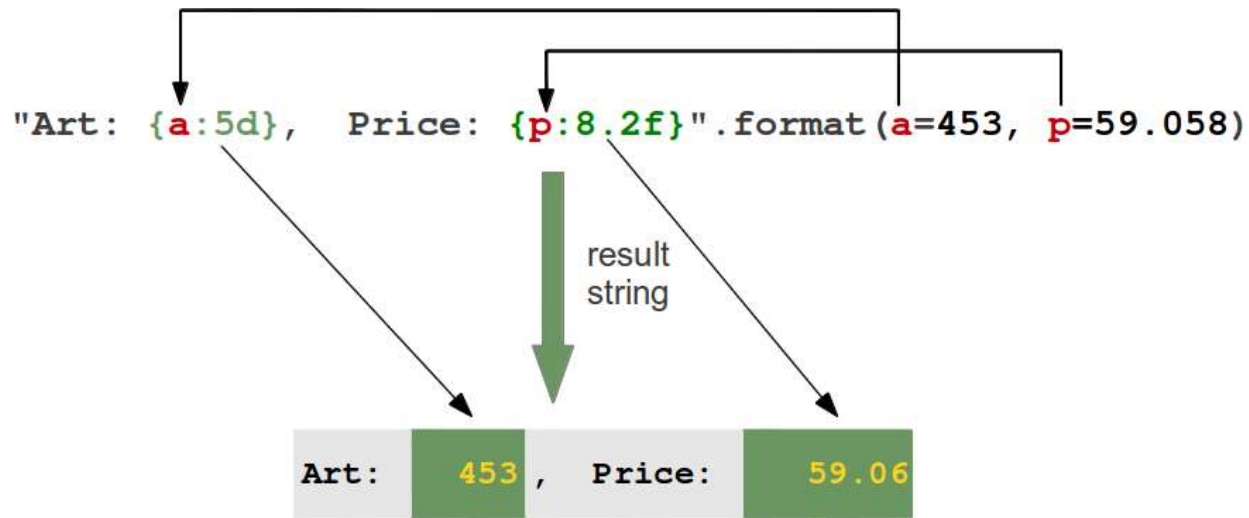
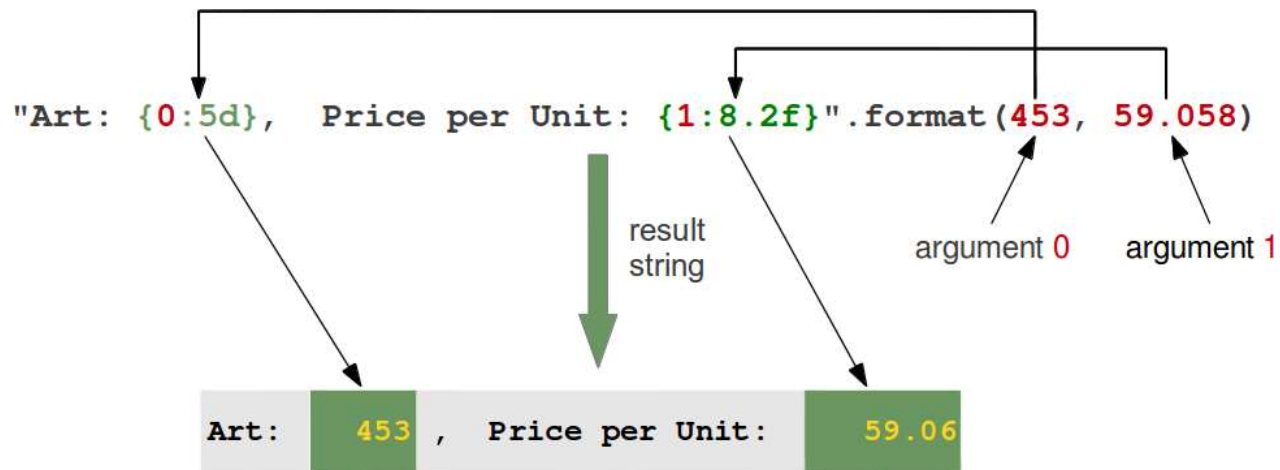
```
print(s) => 'a, b, c'
```

```
s = '{2}, {1}, {0}'.format('a', 'b', 'c')
```

```
print(s) => 'c, b, a'
```



Format Method



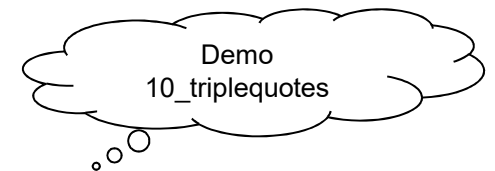
Triple Quotes

- **Allow strings to span multiple lines:**
 - Including verbatim NEWLINEs, TABs, and any other special characters
- **Syntax consists of three consecutive single or double quotes:**

```
para_str = """this string is made up of several lines and non-printable
characters such as TAB ( \t ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like this within the
brackets [ \n ], or just a NEWLINE
within the variable assignment will also show up."""
print(para_str)
```

- **Result:**

```
this string is made up of several lines and non-printable
characters such as TAB (    ) and they will show up that way when
displayed. NEWLINEs within the string, whether explicitly given like
this within the brackets [
], or just a NEWLINE
within the variable assignment will also show up.
```



Raw and Unicode Strings

- **Raw String don't treat backslash as a special character:**
 - Every character in a raw string stays the way it was
- **In a normal String the backslash can be escaped:**

```
print('C:\\nowhere')
```
- **Result:** C:\\nowhere
- **A raw string is made with the `r` in front of the String:**

```
print(r'C:\\nowhere')
```
- **Result:** C:\\nowhere
- **Since Python 3 all strings are stored as Unicode internally:**
 - In Unicode strings consist of (32-bit) code points
 - Allows more characters, including characters from most languages in the world
- **unicodedata module:**
 - Provides access to Unicode Character Database (UCD)
 - UCD contains character properties for all Unicode characters

Sequences

- **Most basic data structure in Python is the sequence:**
 - Each element of a sequence is assigned a number - its position or index
 - First index is zero, the second index is one, and so forth
 - Python distinguishes sequences on the basis of mutability
- **Immutable sequences:**
 - Contain objects that cannot change once created:
 - Objects may references to other objects and these other objects may be changed
 - Examples are Strings (Unicode items), Tuples, Bytes (bytes items)
- **Mutable sequences:**
 - Can be changed after they are created
 - Examples are Lists and Byte Arrays
- **Operations that work on all sequences types:**
 - indexing, slicing, adding, multiplying, and checking for membership
- **Python has built-in functions for:**
 - Finding the length of a sequence and its largest and smallest elements

Lists

- **Lists are mutable sequences and similar to arrays in C:**
 - List contains items separated by commas and enclosed within square brackets []
 - Difference is that list items can be of different data type
- **Values are accessed using the slice operator [] and [:] :**
 - Indexes start at 0, then upwards and with end-1
 - Lists can be sliced, concatenated and so on
- **Lists respond to the + and * operators much like strings:**
 - Meaning is concatenation and repetition
 - Except that result is a new list, not a string
- **Lists respond to all sequence operations usable on strings:**

```
lst = [ 'abcd', 786 , 2.23, 'john', 70.2 ]  
print(lst[0])           # Prints first element of the list  
print(lst[1:3])         # Prints elements starting from 2nd to 4th  
print(lst * 2)          # Prints list two times  
print(lst)              # Prints complete list
```

Accessing and Updating Lists

- **Update single or multiple elements of lists:**
 - Slice list on the left-hand side of the assignment operator

```
l = ['physics', 'chemistry', 1997, 2000]
print("Value available at index 2: ")
print(l[2])    => 1997
l[2] = 2001
print("New value available at index 2: ")
print(l[2])    => 2001
```

- **Elements can be added to a list with `append()`:**

```
l = [1, 2, 3]
l.append(4)
l.append(5)
print(l)      => [1, 2, 3, 4, 5]
```

Multidimensional Lists

- **Multidimensional lists created by using other lists as elements**

```
multilist = [['a', 'b', 'c'], [1, 2, 3, 4]]
```

```
# first dimension
```

```
print(multilist[0][0])
```

```
print(multilist[0][1])
```

```
print(multilist[0][2])
```

```
# second dimension
```

```
print(multilist[1][0])
```

```
print(multilist[1][1])
```

```
print(multilist[1][2])
```

```
print(multilist[1][3])
```

```
# print(multilist[1][4]) # IndexError: list index out of range
```

Delete List Elements

- **Remove a list element with `del` or `remove()`**
- **Use `del` statement if you know index of element to remove:**

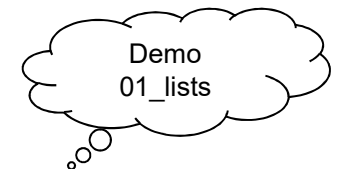
```
l = ['physics', 'chemistry', 1997, 2000]
del l[2]
print("After deleting value at index 2: ")
print(l)    => ['physics', 'chemistry', 2000]
```

- **Use `remove()` if want to remove element by value:**

```
l = ['physics', 'chemistry', 1997, 2000]
l.remove('physics')
print(l)    => ['chemistry', 1997, 2000]
```

- **Delete from a multidimensional list:**

```
multilist = [['a', 'b', 'c'], [1, 2, 3, 4]]
del multilist[0][2]
multilist.remove(['a', 'b', 'c'])
```

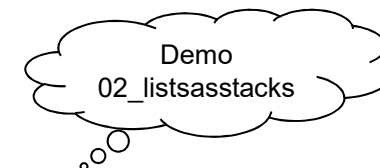


Operations on sequences

| Python Expression | Results | Description |
|--|---------------------------------|---------------|
| <code>len([1, 2, 3])</code> | 3 | Length |
| <code>[1, 2, 3] + [4, 5, 6]</code> | <code>[1, 2, 3, 4, 5, 6]</code> | Concatenation |
| <code>['go'] * 3</code> | <code>['go', 'go', 'go']</code> | Repetition |
| <code>3 in [1, 2, 3]</code> | True | Membership |
| <code>for x in [1, 2, 3]: print x</code> | 1 2 3 | Iteration |

- **indexing and slicing:** `lst = ['one', 'two', 'three']`

| Python Expression | Results | Description |
|----------------------|-------------------------------|--------------------------------|
| <code>lst[2]</code> | <code>'three'</code> | Offsets start at zero |
| <code>lst[-2]</code> | <code>'two'</code> | Negative: count from the right |
| <code>lst[1:]</code> | <code>['two', 'three']</code> | Slicing fetches sections |



Tuples

- **Tuple is a sequence of immutable Python objects:**
 - Consist of a number of values separated by commas
 - Tuples are sequences, just like lists, but tuples are faster because read-only
- **Main differences between lists and tuples:**
 - Lists are enclosed in brackets [] and their elements and size can be changed
 - Tuples are optionally enclosed in parentheses () and cannot be updated
- **Create tuple by assigning comma-separated values to variable:**

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d "
```
- **Empty tuple is written as two parentheses containing nothing:**

```
tup1 = ()
```
- **For tuple containing a single value include a comma:**
 - Otherwise it will not be recognized as tuple

```
tup1 = (50,)    or: tup1 = 50,
```

Accessing Values in Tuples

- **All of the general sequence operations can be applied on tuples:**
 - Use square brackets for slicing along with index to obtain value, indices start at 0
 - Use the + and * operators for concatenation and repetition

```
tup = (1, 2, 3, 4, 5, 6, 7 )  
tup[0] == 1  
tup[1:5] == [2, 3, 4, 5]
```

- **Tuples are immutable and can be thought of as read-only lists:**
 - Cannot update them or change values of tuple elements

```
tuple[2] = 1000    # TypeError
```

- **Portions of existing tuples can be used to create new tuples:**

```
tup1 = (12, 34.56)  
tup2 = ('abc', 'xyz')  
tup3 = tup1 + tup2  
tup3 == (12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

- **Removing individual tuple elements is not possible:**
 - Another tuple with undesired elements discarded must be created
- **Entire tuple can be removed with `del` statement:**

```
tup = ('physics', 'chemistry', 1997, 2000)
print tup    => ('physics', 'chemistry', 1997, 2000)
del tup
print("After deleting tup: ")
print(tup)
```

- **Result after deleting tup:**

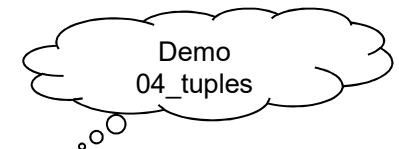
```
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
NameError: name 'tup' is not defined
```

- **Exception since after `del tup` tuple is gone**

Usage of Tuples

- **Tuples have many uses:**
 - Coordinate pairs, records from a database
- **Any set of comma-separated, objects default to tuples:**
 - Written without identifying symbols like brackets or parentheses

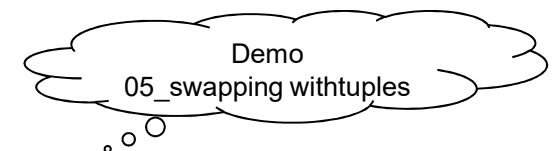
```
print('abc', -4.24e93, 'xyz')    => abc -4.24e+93 xyz
x, y = 1, 2
print("Value of x , y: ", x,y) => Value of x , y: 1 2
```
- **Tuples are faster than lists:**
 - It is faster to loop through a tuple than through a list
- **Tuples can make your code safer:**
 - Allows you to “write-protect” data that does not need to be changed
- **A tuple can have lists as items:**
 - The items in the lists are mutable however



Tuple Functions

- Only functions and methods that do not modify tuple available

| Name | Description |
|---------------------------|---|
| <code>len(tuple)</code> | Gives the total length of the tuple |
| <code>max(tuple)</code> | Returns item from the tuple with max value |
| <code>min(tuple)</code> | Returns item from the tuple with min value |
| <code>tuple(seq)</code> | Converts a sequence into tuple |
| <code>tup.index(i)</code> | index of the first occurrence of <code>i</code> in <code>tup</code> |
| <code>tup.count(i)</code> | total number of occurrences of <code>i</code> in <code>tup</code> |



Bytes and Byte Arrays

- **Bytes:**
 - A bytes object is an immutable array
 - Items are 8-bit bytes, represented by integers in the range $0 \leq x < 256$
- **Bytes objects are constructed with:**
 - Bytes literals like `b'abc'` or the built-in function `bytes()`
 - Bytes objects can be decoded to strings via `decode()`
- **Byte Arrays:**
 - bytearray object is a mutable array
 - Created by built-in `bytearray()` constructor
 - Same interface as immutable bytes objects
- **Extension modules** `array` **and** `collections:`
 - Provide additional mutable sequence example

Sets

- **Data type representing unordered collection with no duplicates**

- **Sets:**

- Mutable set created by `set()` constructor or by using curly braces `{ }`

```
s = {1, 2}
print(s)    => {1, 2}
```

- Can be modified afterwards by several methods like `add()`

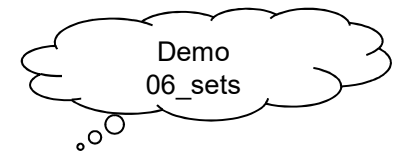
- **Frozen sets:**

- Immutable set created by `frozenset()` constructor

- **Support mathematical operations like union and intersection:**

```
a = set('abracadabra')
b = set('alacazam')
print(a)           # unique letters in a
                   # {'a', 'r', 'b', 'c', 'd'}

print(a - b)       # letters in a but not in b
                   # {'r', 'd', 'b'}
```



Dictionary Characteristics

- **Mutable container type:**
 - Can store any number of Python objects, including other container types
- ```
d = {'Alice': '2341', 18: [1,8], 'Cecil': 3258}
```
- **In literal representation key is separated from value by a colon :**
    - Items (key-value pairs) are separated by commas
    - Notice curly braces
    - Empty is written with just two curly braces: { }
  - **Keys are unique within a dictionary while values need not be**
  - **Values of a dictionary can be of any type**
  - **Keys must be immutable objects such as strings, numbers or tuples**
  - **Items are unordered**

## Accessing Values in Dictionary

- **Use square brackets along with the key to obtain its value:**

```
d = {'Name': 'Albert', 'Age': 55, 'Class': 'First'}
print("d['Name']: ", d['Name']) => d['Name']: Albert
print("d['Age']: ", d['Age']) => d['Age']: 55
```

- **Access data item with key not in dictionary gives error:**

```
d = {'Name': 'Albert', 'Age': 55, 'Class': 'First'}
print("looking up 'Robert': ", d['Robert'])
```

- **Result:**

```
d['Robert']:
Traceback (most recent call last):
 File "test.py", line 4, in <module>
 print "d['Robert']: ", d['Robert']
KeyError: 'Robert'
```

# Updating Dictionaries

- **A dictionary can be updated by:**
  - Adding a new entry or item (i.e., a key-value pair)
  - Modifying an existing entry
  - Deleting an existing entry

```
d = {'Name': 'Albert', 'Age': 55, 'Class': 'First'}
d['Age'] = 56 # Update existing entry
d['University'] = "Zurich"; # Add new entry
print("d['Age']: ", d['Age'])
print("d['University']: ", d['University'])
```

- **Result:**

```
d['Age']: 56
d['University']: Zurich
```

# Delete Dictionary Elements

- **Several options to delete with `del` statement are available:**

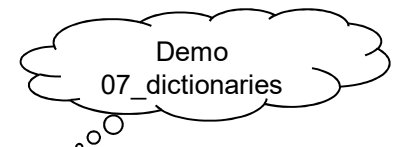
- Individual dictionary elements can be removed
- Entire contents of a dictionary can be cleared with `clear()`
- Entire dictionary in single operation with `del`

```
d = {'Name': 'Albert', 'Age': 55, 'Class': 'First'}
del d['Name'] # remove entry with key 'Name'
d.clear() # remove all entries in d
del d # delete entire dictionary
print("d['Age']: ", d['Age'])
```

- **Result :**

```
d['Age']:
Traceback (most recent call last):
 File "test.py", line 8, in <module>
 print "d['Age']: ", d['Age']
TypeError: 'type' object is unsubscriptable
```

- **Exception because `d` dictionary does not exist**



## Properties of Dictionary Keys

- **Dictionary values have no restrictions:**
  - Can be any Python object
- **There two important points to about dictionary keys:**
  - No duplicate keys allowed
  - Keys must be immutable
- **In case duplicate keys are encountered during assignment:**
  - Last assignment wins

```
>>> d = {'Name': 'Albert', 'Age': 7, 'Name': 'Robert'}
```

```
>>> d['Name']
```

```
Robert
```

## Non Mutable Keys

- **Strings, numbers, or tuples can be dictionary keys:**
  - But something like `['Name']` is not allowed since a list is mutable

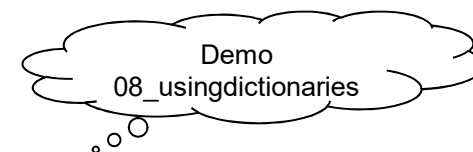
```
d = {'Name': 'Albert', 'Age': 55}
print("d['Name']: ", d['Name'])
```

- **Result:**

```
Traceback (most recent call last):
 File "test.py", line 3, in <module>
 d = {'Name': 'Albert', 'Age': 55}
TypeError: list objects are unhashable
```

# Dictionary Functions and Methods

| Name                           | Description                                                    |
|--------------------------------|----------------------------------------------------------------|
| <code>len(dict)</code>         | Gives the total length or nr. of items in the dictionary.      |
| <code>str(dict)</code>         | Produces a printable string representation of a dictionary     |
| <code>fromkeys()</code>        | Create new dictionary with keys from seq , values set to value |
| <code>dict.clear()</code>      | Removes all elements of dictionary dict                        |
| <code>dict.copy()</code>       | Returns a shallow copy of dictionary dict                      |
| <code>dict.get(key)</code>     | For key key, returns value or default if key not in dictionary |
| <code>dict.popitem()</code>    | Remove and return arbitrary (key, value) pair from dictionary  |
| <code>dict.items()</code>      | Returns a list of dict's (key, value) tuple pairs              |
| <code>dict.keys()</code>       | Returns list of dictionary dict's keys                         |
| <code>dict.update(dict)</code> | Adds dictionary dict's key-values pairs to dict                |
| <code>dict.values()</code>     | Returns list of dictionary dict's values                       |



# Control Flow Constructs

- **Every programming has control flow constructs:**
  - Sequential constructs like statements
  - Branching constructs to take a certain direction
  - Iteration construct to repeat code a number of times
- **Branching Statements:**
  - `if`
  - `if...else`
  - `elif`
- **Iteration Statements:**
  - `for`
  - `while`
  - `continue`
  - `pass`



## if Statement

- **if statement in Python is similar to that of other languages:**
  - Contains a logical expression using which data is compared
  - Decision is made based on the result of the comparison
- **Syntax if statement:**

```
if expression:
 statement(s)
```
- **Expression is evaluated first:**
  - If true that is, if its value is nonzero
  - Then statement(s) block are executed
  - Otherwise next statement following statement(s) block is executed
- **Grouping statements:**
  - Statements indented by the same number of character spaces:
    - Considered to be part of a single block of code.
  - Python uses indentation as its method of grouping statements

## if Example

```
var1 = 100
if var1:
 print("1 - Got a true expression value")
 print(var1)
```

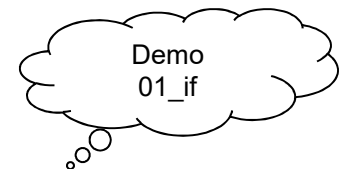
```
var2 = 0
if var2:
 print("2 - Got a true expression value")
 print var2
print("Good bye!")
```

- **Result:**

```
1 - Got a true expression value
100
Good bye!
```

- **In case if has only one line it may go on same line:**

```
if expression == 1: print("Value of expression is 1")
```

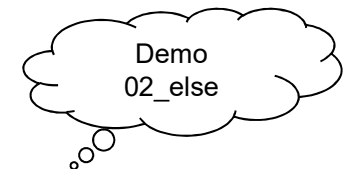


## else Statement

- **Is combined with an `if` statement:**
  - Contains code that executes in case the `if` statement resolves to 0 or false
- **`else` statement is an optional statement:**
  - Only one `else` statement can follow the `if` statement
- **Syntax `if...else` statement:**

```
if expression:
 statement(s)

else:
 statement(s)
```

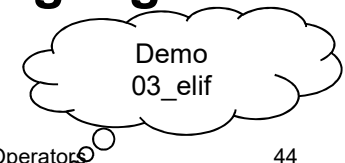


## elif Statement

- **Allows checking of multiple expressions for truth value:**
  - Execute a block of code as soon as one of the conditions evaluates to true
- `elif` **statement is optional:**
  - Arbitrary number of `elif` statements may be following an `if`
- **Syntax of the `if...elif` statement:**

```
if expression1:
 statement(s)
elif expression2:
 statement(s)
elif expression3:
 statement(s)
else:
 statement(s)
```

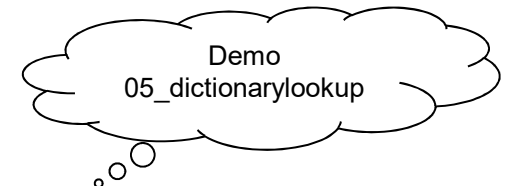
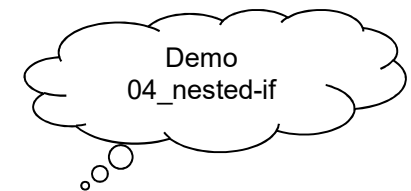
- **No support for `switch` or `case` statements as in other languages**



## Nested if...elif...else

- `if...elif...else` **construct in other** `if...elif...else` **construct:**
  - Allows checking another condition after condition resolves to true
- **Syntax nested** `if...elif...else:`

```
if expression1:
 statement(s)
 if expression2:
 statement(s)
 elif expression3:
 statement(s)
 else:
 statement(s)
elif expression4:
 statement(s)
else:
 statement(s)
```



# while Loop

- **Control flow construct to repeat code:**

- Repetition continues until conditional expression becomes false
- condition must be a logical expression evaluating to true or false

```
while conditional expression:
 statement(s)
```

- **while loop ends when conditional expression becomes false**

- **infinite loop when condition of a while never resolves to false:**

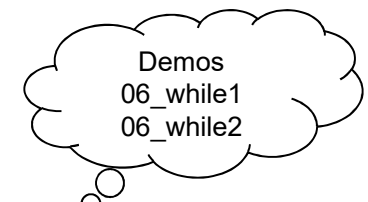
- Infinite loop might be useful in client/server programming:

Server runs continuously on an open port so that clients can communicate with it

- **If while clause has only one statement:**

- Placement on the same line is allowed

```
while expression: statement
```

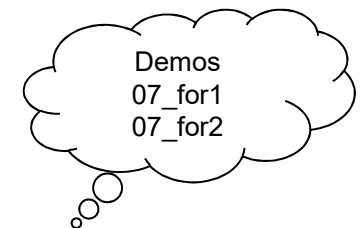


# for Loop

- **Control flow construct to repeat code number of times:**
  - Iterates over the items of an iterator (any sequence, such as a list or a string)

```
for iterating_var in sequence:
 statements(s)
```

- **for loop operation:**
  - First item in the sequence is assigned to iterating variable `iterating_var`
  - Next statements block is executed
  - Each item in the sequence is assigned to `iterating_var`
  - Statements(s) block is executed until the entire sequence is finished



## break and continue Statements

- `break` **statement:**
  - Terminates the current loop iteration
  - Resumes execution at next statement after the loop
- `continue` **statement returns control to the beginning of the loop:**
  - Remaining statements in the current iteration of the loop are skipped
  - Control moves back to the top of the loop
- `break` **and** `continue` **can be used in both** `while` **and** `for` **loops**
- **Example** `continue` **statement:**

```
for letter in 'Python':
 if letter == 'h':
 continue
 print('Current Letter:', letter)
print("Good bye!")
```

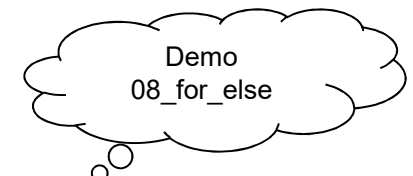
- **In result letter `h` is not printed**



## Loop with else Combination

- `else` **statement can be associated with loop statements**
- **If `else` statement is used with a `for` loop:**
  - `else` statement is executed when the loop has exhausted iterating the list.
- **If `else` statement is used with a `while` loop:**
  - `else` statement is executed when the condition becomes false
- **In the example on the next slide:**
  - `else` statement is combined with of a `for` statement
  - Search is done for prime numbers from 10 through 20

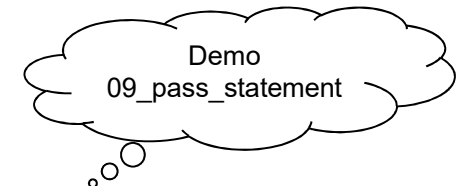
```
for num in range(10,20): # to iterate between 10 to 20
 for i in range(2,num): # to iterate on the factors of the number
 if num%i == 0: # has factor, cannot be prime
 j=num/i
 print('%d equals %d * %d' % (num,i,j))
 break # to move to the next number, the #first FOR
 else: # else part of the loop
 print(num, 'is a prime number')
```



## pass Statement

- **Used when a statement is required syntactically:**
  - But it is not desired that any command or code executes
- `pass` **statement is a null operation:**
  - Nothing happens when it executes
  - Also useful in places where code has not been written yet (e.g. in stubs )
- `pass` **statement is helpful when:**
  - A code block is created but it is no longer required
  - Statements inside the block may then be removed
  - Block may remain with pass statement so it doesn't interfere with other code parts

```
for letter in 'Python':
 if letter == 'h':
 pass
 print('This is pass block')
 print('Current Letter: ', letter)
print("Good bye!")
```



# Python Arithmetic and Assignment Operators

| Operator | Arithmetic Operator Description                                       |
|----------|-----------------------------------------------------------------------|
| +        | Adds values on either side of the operator                            |
| -        | Subtracts right hand operand from left hand operand                   |
| *        | Multiplies values on either side of the operator                      |
| /        | Divide left hand operand by right hand operand with decimal result    |
| //       | Division with quotient result with digits after decimal point removed |
| %        | Divides left hand operand by right hand operand, returns remainder    |
| **       | Performs exponential (power) calculation on operators                 |

| Operator | Assignment Operator Description                                               |
|----------|-------------------------------------------------------------------------------|
| =        | Assigns values from right side operands to left side operand                  |
| +=       | Adds right operand to left operand and assigns result to left operand         |
| -=       | Subtracts right operand from left operand and assigns result to left operand  |
| *=       | Multiplies right operand with left operand and assigns result to left operand |
| /=       | Divides left operand by right operand and assigns result to left operand      |
| %=       | Takes modulus using two operands and assigns result to left operand           |

# Python Comparison and Bitwise Operators

| Operator | Comparison Operator Description                                             |
|----------|-----------------------------------------------------------------------------|
| ==       | Checks if values of two operands are equal or not. If yes condition is true |
| !=       | Checks if values of two operands are equal or not. If not condition is true |
| <>       | Same as != operator                                                         |
| >        | Checks if value of left operand is greater than right operand               |
| <        | Checks if value of left operand is less than right operand                  |
| >=       | Checks if value of left operand is greater than or equal to right operand   |
| <=       | Checks if value of left operand is less than or equal to right operand      |

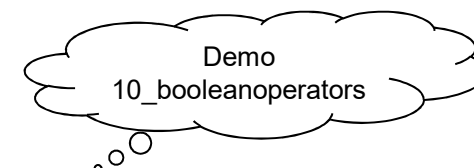
| Operator | Bitwise Operator Description                                                 |
|----------|------------------------------------------------------------------------------|
| &        | Copies a bit to the result if it exists in both operands                     |
|          | Copies a bit to the result if it exists in one or both of the operands       |
| ^        | Copies a bit to the result if it is set in one but not both operands         |
| ~        | Unary Binary Ones Complement Operator. Has effect of flipping the bits       |
| <<       | Left operands value is moved left by nr. of bits specified by right operand  |
| >>       | Left operands value is moved right by nr. of bits specified by right operand |

# Logical, Membership and Identity Operators

| Operator         | Logical Operator Description                                        |
|------------------|---------------------------------------------------------------------|
| <code>and</code> | If both operands are true then the condition becomes true           |
| <code>or</code>  | If any of the operands are non-zero then the condition becomes true |
| <code>not</code> | Reverses logical state of operand. True condition will become false |

| Operator            | Membership Operator Description                                               |
|---------------------|-------------------------------------------------------------------------------|
| <code>in</code>     | Evaluates to true if it finds variable in specified sequence, false otherwise |
| <code>not in</code> | Evaluates to true if it does not find a variable in sequence, false otherwise |

| Operator            | Identity Operator Description                                                |
|---------------------|------------------------------------------------------------------------------|
| <code>is</code>     | Evaluates to true if variables on both sides point to the same object        |
| <code>is not</code> | Evaluates to true if variables on both sides do not point to the same object |



# Function Syntax

- **First statement of a function can be an optional statement:**

- Documentation string of the function or `docstring`

```
def functionname(parameters):
 "function_docstring"
 function_suite
 return [expression]
```

- **By default:**

- Parameters have a positional behavior
- Pass them in the same order that they were defined

- **Example of a simple Python function:**

- Takes a string as input parameter and prints it on standard screen

```
def printme(s):
 "This prints a passed string into this function"
 print(s)
 return # redundant
```

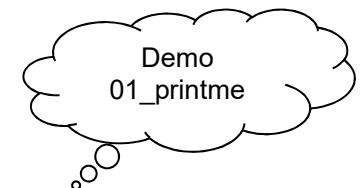
# Calling Functions

- **Execute functions by calling them:**
  - From another function or directly from the Python prompt
  - If functions specify parameters, it is required to pass them
- **Function must be defined before they can be called:**
  - Order of definition and call is important
- **Example of calling the previously defined `printme()` function:**

```
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
printme() # results in error
```

- **Result:**

```
I'm first call to user defined function!
Again second call to the same function
Traceback (most recent call last):
 File "01_printme.py", line 8, in <module>
 printme()
TypeError: printme() takes exactly 1 argument (0 given)
```



## Pass by sharing

- **Python passes function parameters by value (i.e. by object):**

- Parameter in function is bound to same object as referred to by variable used in call
- Change to object via parameter in function is seen by variable in function's caller
- So changes to mutable objects made by function will also affect variable in call

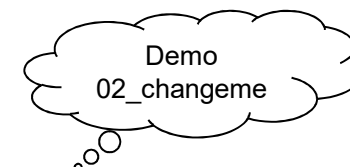
```
def changeme(mylist):
 mylist.append([1,2,3])
 print("Values inside the function: ", mylist)
 return
```

- **Call to changeme function:**

```
mylistouter = [10,20,30]
changeme(mylistouter)
print("Values outside the function: ", mylistouter)
```

- **Reference of passed object is used to append values:**

```
Values inside function: [10, 20, 30, [1, 2, 3]]
Values outside function: [10, 20, 30, [1, 2, 3]]
```





# Overwriting parameters

- **Parameter may be overwritten in the function:**

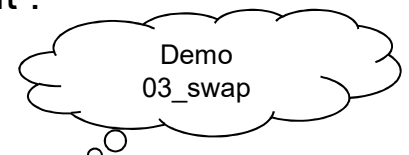
```
def changeme(mylist):
 mylist = [1,2,3,4] # This would assign new reference
 print("Values inside the function: ", mylist)
 return
```

- **Call** changeme **function:**

```
mylistouter = [10,20,30]
changeme(mylistouter)
print("Values outside the function: ", mylistouter)
```

- **Parameter** mylist **is changed in the function** changeme:
  - Now changing mylist within the function does not affect mylistouter
  - Function does nothing and finally this would produce following result :

```
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
```



- **Changing immutable references always results in new objects:**
  - Is true for Strings and Numbers

# Keyword Arguments

- **Function definition:**

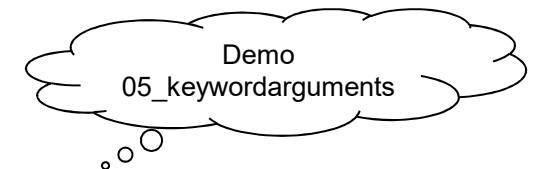
```
def printinfo(name, age):
 "This prints name and age passed into this function"
 print("Name:", name)
 print("Age", age)
 return
```

- **Function call with keyword arguments:**

```
printinfo(age=50, name="miki")
```

- **Result:**

```
Name: miki
Age 50
```



# Default Arguments

- **Function definition:**

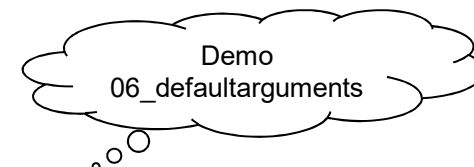
```
def printinfo(name, age = 35):
 "This prints name and age passed into this function"
 print("Name: ", name)
 print("Age ", age)
 return
```

- **Functions calls with and without default arguments:**

```
printinfo(age=50, name="miki")
printinfo(name="miki")
```

- **Results:**

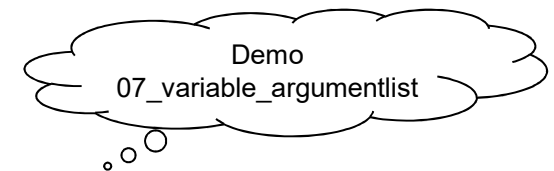
```
Name: miki
Age 50
Name: miki
Age 35
```



# Variable-length arguments

```
def functionname([args,] *var_args_tuple, **var_args_keywords):
 "function_docstring"
 function_suite
 return [expression]
```

- `*var_args_tuple`:
  - Asterisk (\*) indicates variable holding values of all non-keyword variable arguments
  - Tuple remains empty if no additional arguments are specified during function call
- `**var_args_keywords`:
  - Double asterisk (\*\*) indicates a variable-length argument list with keywords
  - Dictionary remains empty if no keyword arguments are passed



# Anonymous Functions

- `lambda` **keyword is used to create small anonymous functions:**
  - Not declared in the standard manner by using the `def` keyword
- **Lambda form can take any number of arguments:**
  - Return just one value in the form of an expression
  - Cannot contain commands or multiple expressions
- **Characteristics of lambda functions:**
  - Have their own local namespace
  - Can only access variables in parameter list and in global namespace
  - A single expression as body

# Syntax Lambda Functions

- **Lambda functions contain only a single statement:**

```
lambda [arg1 [,arg2,.....argn]]:expression
```

- **Function definition:**

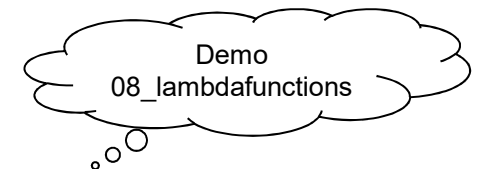
```
sum = lambda arg1, arg2: arg1 + arg2
```

- **Function call:**

```
print("Value of total: ", sum(10, 20))
print("Value of total: ", sum(20, 20))
```

- **Result:**

```
Value of total: 30
Value of total: 40
```



# return Statement

- `return [expression]` **exits a function:**
  - Optionally passing back an expression to the caller
  - `return` statement with no arguments is the same as `return None`

- **Function definition:**

```
def sum(arg1, arg2):
 # Add both the parameters and return them."
 total = arg1 + arg2
 print("Inside the function: ", total)
 return total
```

- **Function call:**

```
total = sum(10, 20);
print("Outside the function: ", total)
```

- **Result:**

```
Inside the function: 30
Outside the function: 30
```

# Scope of Variables

- **Accessibility of variable depends on where it is declared:**
  - Scope determines where in program you can access a particular identifier
- **There are two basic scopes of variables in Python:**
  - Global variables
  - Local variables
- **Global versus Local variables:**
  - Variables defined inside a function body have a local scope
  - Variables defined outside have a global scope
- **Local variables:**
  - Can be accessed only inside the function in which they are declared
  - When you call a function, the variables declared inside it are brought into scope
- **Global variables:**
  - Can be accessed throughout the program body by all functions
  - Refer to global instead of a local variable inside function with `global` keyword



## Example Variable Scope

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

- **Function definition:**

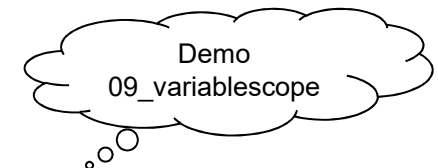
```
def sum(arg1, arg2):
 # Add both the parameters and return them."
 total = arg1 + arg2; # Here total is local variable.
 print("Inside the function local total: ", total)
 return total;
```

- **Function call:**

```
sum(10, 20)
print("Outside the function global total: ", total)
```

- **Result:**

```
Inside the function local total: 30
Outside the function global total: 0
```



# Modules

- **Used to organize number of Python definitions in single file:**
  - Used to logically organize your Python code
  - Can be bind and referenced
- **Module is a file consisting of Python code:**
  - Defines functions, classes, and variables
  - May also include runnable code
  - Grouping related code into a module makes code easier to understand and use
- **Example:**
  - Python code for a module named test normally resides in a file named `test.py`
- `test.py`:

```
def print_func(par):
 print("Hello: ", par)
```

# import Statement

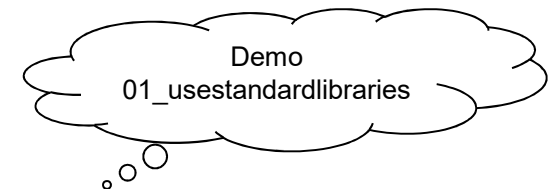
- **Any Python source file can be used as a module:**
  - By executing an `import` statement in some other Python source file
- `import` **has the following syntax:**

```
import module1[, module2[, ... moduleN]
```
- **On encountering an `import` statement:**
  - Module is imported if the module is present in the search path
  - Search path is list of directories that interpreter searches for modules to import
  - A module is loaded only once, regardless of the number of times it is imported
- **Example:**
  - To import module `test.py`, put the following command at the top of the script:

```
import test # Now you can call defined function
test.print_func('pipo') => Hello: pipo
```

- **Name module can be changed with:**

```
import test as t
t.print_func('pipo') => Hello: pipo
```



## from...import Statement

- Imports specific attributes from module into current namespace:

```
from modname import name1[, name2[, ... nameN]]
```

- Function `sqrt` from module `math` is imported with:

```
from math import sqrt
```

- Statement does not import the entire module `math`:

- Only item `sqrt` from module `math` in global symbol table of importing module

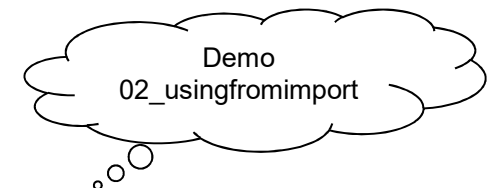
- Now the function `sqrt` can be used without prefix:

```
print(sqrt(9)) => 3.0
```

- Also possible is to import all names from a module with:

```
from modname import *
```

- Statement should be used sparingly



## Locating Modules

- **On import Python interpreter searches for module as follows:**
  - Current directory
  - Each directory in the shell variable `PYTHONPATH`
  - Default path which is OS dependent
- **Search path is stored in system module `sys` as `sys.path` variable**
- **`PYTHONPATH` variable:**
  - Environment variable, consisting of a list of directories
  - Syntax of `PYTHONPATH` is same as that of the shell variable `PATH`
- **Typical `PYTHONPATH` from Windows system:**

```
set PYTHONPATH=c:\python20\lib;
```
- **Typical `PYTHONPATH` from UNIX system:**

```
set PYTHONPATH=/usr/local/lib/python
```

# Creating and Using Modules

- **Every Python program is already a module:**

- Modules must have `.py` extension

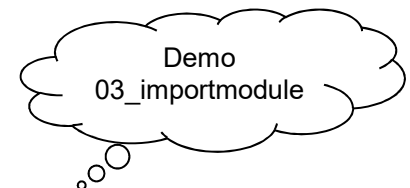
```
Filename: mymodule.py
def sayhi():
 print('Hi, this is mymodule speaking.')
__version__ = '0.1'
End of mymodule.py
```

- **Module can be imported by other module if on `sys.path`:**

```
import mymodule
mymodule.sayhi()
print('Version', mymodule.__version__)
```

- **Alternatively `from .. import` can be used:**

```
from mymodule import sayhi, __version__
sayhi()
print('Version', __version__)
```



# dir Function

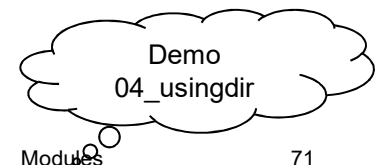
- **Returns sorted list of strings with names defined by a module**
- **List contains the names defined in a module:**
  - Modules, Variables, Functions
- **Example** `import built-in module math:`

```
import math
content = dir(math)
print(content)
```

- **Result:**

```
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
```

- `__name__` **is module's name**
- `__package__` **is package name of module**



# Packages in Python

- **Package is way to organize number of modules together as unit:**
  - Has a hierarchical file directory structure
  - Consists of modules and subpackages and sub-subpackages, and so on
- **Multiple functions in files and different Python classes:**
  - Create packages out of those classes
- **Typically modules of a package are placed in a directory:**
  - Consider for example `Canon.py`, `Leika.py` and `Nikon.py` in `cameras` directory
- `Canon.py` **could define a simple function like:**

```
def CanonInfo():
 print("Canon camera")
```
- **Other two files could define similar but different functions:**
  - `def Leika()` and `def Nikon()`
- **Package can have `__init__.py` in its directory (must < python3.3):**
  - Makes functions available when package is imported



# Explicit Import of Package Modules

- **modules of a package can be imported explicitly:**

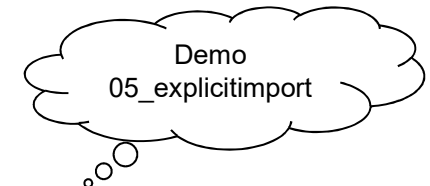
```
import cameras.Canon
import cameras.Leika
import cameras.Nikon
```

- **Functions in package are called with full name:**

```
cameras.Nikon.NikonInfo()
cameras.Canon.CanonInfo()
cameras.Leika.LeikaInfo()
```

- **Full name consists of:**

- Package name (directory), module name, function name



## Implicit Import of Package Modules

- **Following lines can be added to `__init__.py` in package :**

```
from cameras.Canon import CanonInfo
from cameras.Leika import LeikaInfo
```

- **Now modules can be imported by importing the package:**

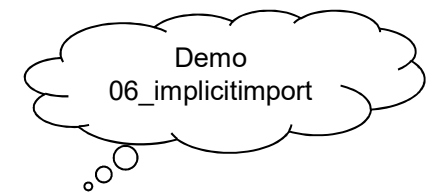
```
import cameras
```

- **Import `cameras` package and call methods:**

```
cameras.CanonInfo()
cameras.LeikaInfo()
```

- **Also `from .. import` can be used:**

```
from cameras import NikonInfo, CanonInfo, LeikaInfo
NikonInfo()
CanonInfo()
LeikaInfo()
```



## globals and locals Functions

- **Used to return the names in the global and local namespaces**
- `locals()` **called from within a function:**
  - Return all the names that can be accessed locally from that function
- `globals()` **called from within a function:**
  - Return all the names that can be accessed globally from that function
- **Return type of both these functions is dictionary:**
  - Names can be extracted using the `keys()` function

## reload Function

- **imports a previously imported module again:**
  - When module is imported, code in the top-level portion is executed only once
- **To reexecute top-level code in a module:**
  - Use the `reload()` function from `importlib` module
  - In previous version `imp` module

- **Syntax:**

```
reload(module_name)
```

- `module_name`:
  - Name of the module you want to reload
  - Not the string containing the module name
- **To re-load hello module:**

```
reload(hello)
```

# Namespaces and Scoping

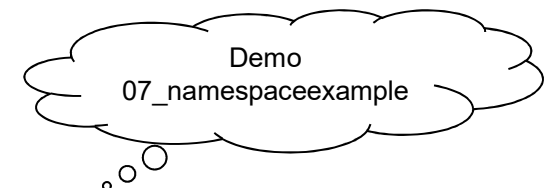
- **Namespace is a dictionary containing:**
  - Variable names (keys) and their corresponding objects (values)
- **Variables can be in local namespace or global namespace:**
  - Variables declared outside functions are in the global namespace
  - Variables declared in function are in the local namespace of that function
  - Class methods follow the same scoping rule as ordinary functions
- **If local and global variable have the same name:**
  - Local variable shadows the global variable
- **Python assumes whether variables are local or global:**
  - Any variable assigned a value in a function is local
  - Use the `global` statement to assign a value to a global variable within a function
- `global VarName` **tells Python that `VarName` is a global variable:**
  - Python stops searching the local namespace for the variable

## Example Namespaces

- **Define variable `Money` in the global namespace:**
  - Assign `Money` a value within the function `AddMoney`
  - Python assumes `Money` is a local variable
- **If value of local variable `Money` is accessed before setting it:**
  - `UnboundLocalError` is the result
  - Uncommenting the global statement fixes the problem

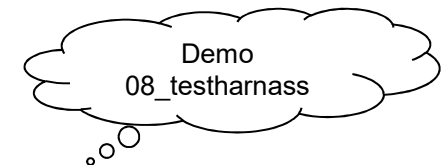
```
Money = 2000
def AddMoney():
 # Uncomment the following line to fix the code:
 # global Money
 Money = Money + 1

print(Money)
AddMoney()
print(Money)
```



# Test Harnass

- **To run code only when script is run, not when imported**
- **`if __name__ == '__main__':`**
  - Only true only when the file is run
  - Not when the module is imported
  - Good practice to test how module is used
- **Code in module only evaluated first time it is imported:**
  - Python maintains an internal list of all modules that have been imported
- **When you import a module for the first time:**
  - Module script is executed in its own namespace until the end
  - Internal list is updated, and execution of continues after the import statement
- **Can use reload to force new import e.g. when module changed**



# Installing Packages

- **From PyPI or from VCS, other indexes, local archives**
- **On Linux or OS X:**
  - `pip install -U pip setuptools`
- **On Windows:**
  - `python -m pip install -U pip setuptools`
- **Common usage of `pip`:**
  - Install from the Python Package Index using a requirement specifier

```
pip install package
pip install package==1.3
pip install package~=1.3
pip install package>1.3
pip install --upgrade package
pip install -r requirements.txt
pip freeze > requirements.txt
pip list
pip uninstall package
```



# Virtual Environments

- **Install packages in separate location, not shared globally**
- **Each environment has its own Python binary**
- **Handles different apps requiring different versions of package**
- **Applications won't break by subsequent library updates**
- **Makes it easier to create distributions**
- **Create virtual environment as follows:**

```
pip install virtualenv
virtualenv c:\path\to\myenv
```

- **OR:**

```
python -m venv myenv c:\path\to\myenv (>Python3.4)
```

- **To make using virtual environments even easier:**

```
pip install virtualenvwrapper
```

# Switching Environments

- `activate` **to set path and change prompt**
- `deactivate` **to return to standard python environment**

| Platform | Shell      | Command to activate virtual environment |
|----------|------------|-----------------------------------------|
| Posix    | bash/zsh   | \$ source <venv>/bin/activate           |
|          | fish       | \$ . <venv>/bin/activate.fish           |
|          | csH/tcsh   | \$ source <venv>/bin/activate.csh       |
| Windows  | cmd.exe    | C:\> <venv>\Scripts\activate.bat        |
|          | PowerShell | PS C:\> <venv>\Scripts\Activate.ps1     |

- `Virtualenvwrapper` **makes switching environments easier:**  
`mkvirtualenv env`  
`workon`  
`rmvirtualenv`

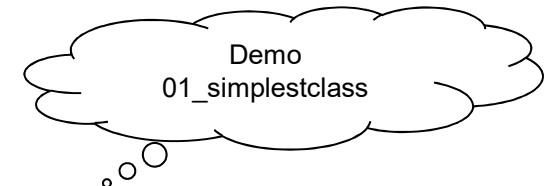
## Creating classes (i.e. data types)

- `class` **statement creates a new class definition:**

```
class ClassName:
 'Optional class documentation string'

 class_suite
```

- **The class has a documentation string:**
  - Can be access via `ClassName.__doc__`
- `class_suite` **consists of statements, executed when defined:**
  - definitions of functions in namespace of class (**methods**)
  - assignment of class attributes

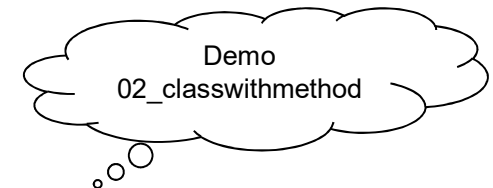


## Example Creating Classes and Objects

```
class Employee:
 'Common base class for all employees'
 empCount = 0
 def __init__(self, name, salary):
 self.name = name
 self.salary = salary
 Employee.empCount += 1
 def displayCount(self):
 print("Total Employee %d" % Employee.empCount)
 def displayEmployee(self):
 print("Name: ", self.name, ", Salary: ", self.salary)
```

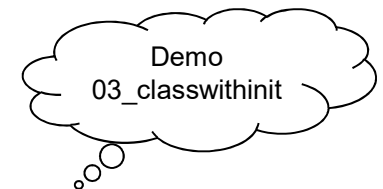
- **To create instances of a class:**
  - Call class using class name and pass arguments that `__init__` method accepts
- **Create object of `Employee` class:**

```
emp1 = Employee("Zara", 2000)
```



# Class Members

- `empCount` :
  - Class variable whose value would be shared among all instances of a this class
  - Accessible as `Employee.empCount` from inside the class or outside the class
- `__init__()` :
  - Special method which is called class constructor or initialization method
  - Called by Python when you create a new instance of this class
- **Instance methods:**
  - Declared like normal functions but first argument to each method is `self`
  - No need to include this argument when you call methods since Python adds it
- `self` **must be explicitly listed as first argument to method:**
  - Instance variables are referred to with "`self.XXX`"
- `self` **is a reference to the instance:**
  - Reference to the container for state of object
  - In Python, instance is visible and *explicit* in method definitions



# Creating and Using Objects

- **Creating objects:**

```
"This would create first object of Employee class"
```

```
emp1 = Employee("Zara", 2000)
```

```
"This would create second object of Employee class"
```

```
emp2 = Employee("Manni", 5000)
```

- **Access object's attributes using dot operator with object :**

```
emp1.displayEmployee()
```

```
emp2.displayEmployee()
```

- **Access class variable using class name:**

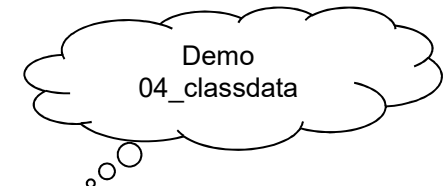
```
print("Total Employee %d" % Employee.empCount)
```

- **Result:**

```
Name: Zara ,Salary: 2000
```

```
Name: Manni ,Salary: 5000
```

```
Total Employee 2
```



# Accessing Attributes

- **In Python can add, remove, or modify attributes at any time:**

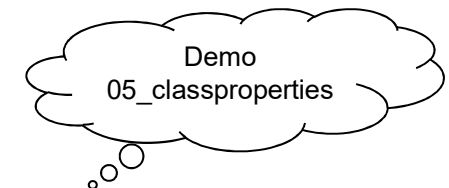
```
emp1.age = 7 # Add an 'age' attribute
emp1.age = 8 # Modify 'age' attribute
del emp1.age # Delete 'age' attribute
```

- **There are alternative functions to access attributes:**

- `getattr(obj, name[, default])`: Access the attribute of object
- `hasattr(obj, name)`: Check if attribute exists or not
- `setattr(obj, name, value)`: Set or create attribute
- `delattr(obj, name)`: Delete an attribute

- **Example code:**

```
hasattr(emp1, 'age') # Returns true if 'age' attribute exists
getattr(emp1, 'age') # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(emp1, 'age') # Delete attribute 'age'
```



## Built-In Class Attributes

- **Every Python class has following built-in attributes:**
  - `__dict__` : Dictionary containing the class's namespace
  - `__doc__` : Class documentation string, or None if undefined
  - `__name__` : Class name.
  - `__module__` : Module name in which the class is defined  
This attribute is "`__main__`" in interactive mode.
  - `__bases__` : A possibly empty tuple containing the base classes  
In the order of their occurrence in the base class list
- **Can be accessed using dot operator like any other attribute**

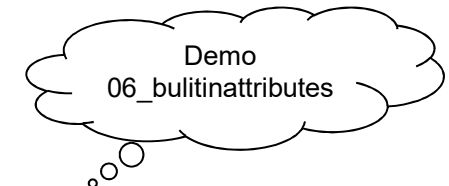


## Example Accessing Built-in Class Attributes

```
print("Employee.__doc__:", Employee.__doc__)
print("Employee.__name__:", Employee.__name__)
print("Employee.__module__:", Employee.__module__)
print("Employee.__bases__:", Employee.__bases__)
print("Employee.__dict__:", Employee.__dict__)
```

- **Result:**

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```



# Destroying Objects

- **Python deletes unneeded objects automatically to free space:**
  - This process of periodically reclaiming memory is termed Garbage Collection
- **Python's garbage collector runs during program execution:**
  - Objects deleted when its reference count reaches zero
- **Reference count changes as references to it change:**
  - Increase when it's assigned a new name
  - Increase when placed in a container (list, tuple, or dictionary)
  - Decrease when var deleted, or removed from collection, with `del`
  - Decrease when reference is reassigned or goes out of scope
- **`__del__()` method:**
  - called when object is actually destroyed
  - may be used to clean up resources used by an instance, for logging etc.

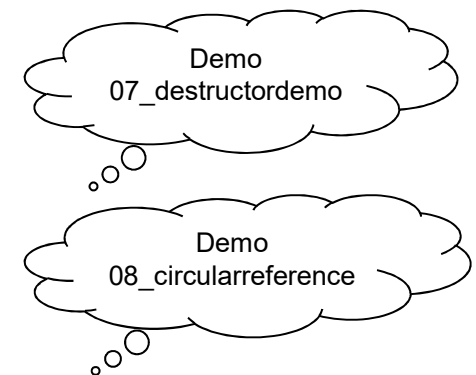
## Example Destructor

```
class Point:
 def __init__(self, x=0, y=0):
 self.x = x
 self.y = y
 def __del__(self):
 class_name = self.__class__.__name__
 print(class_name, "destroyed")

pt1 = Point()
pt2 = pt1
pt3 = pt1
print(id(pt1), id(pt2), id(pt3)) # prints id's of objects
del pt1
del pt2
del pt3
```

- Result:**

```
3083401324 3083401324 3083401324
Point destroyed
```



## Data Hiding

- **Python does not support real data hiding as other languages:**
  - Attributes and methods can however be made more difficult to access
- **Attributes made private with double underscore prefix in name:**

```
__hidden = 0
```

- **Attributes (methods) will not be directly visible to outsiders:**
  - Access of such attributes from outside should include class name

```
instance._InstanceClassName__hidden
```

# Class Inheritance

- **Class can be derived from one or more parent classes:**
  - List parent classes in parentheses after the class name
- **Child class inherits the attributes of its parent class:**
  - Attributes can be used as if they were defined in the child class
  - Child class can also override data members and methods from the parent
- **Derived classes are declared much like their parent class:**

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
 'Optional class documentation string'
 class_suite
```

- **Subclasses can initialize base class object by:**
  - Calling `super().__init__()`
  - Calling `BaseClass.__init__(self, name)`

## Example Class Derivation

```
class Employee: # define base class
 def __init__(self, name):
 self.name = name
 def calcSalary(self):
 self.salary = 0
 return self.salary
 def setName(self, name): self.name = name
 def getName(self): return self.name

class WageEmployee(Employee): # define derived class
 def __init__(self, name, wage, hours):
 super().__init__(name)
 # Employee.__init__(self, name) # alternative
 self.wage = wage
 self.hours = hours
 def setWage(self, wage): self.wage = wage
 def getWage(self): return self.wage
```

## Example Using Derived Classes

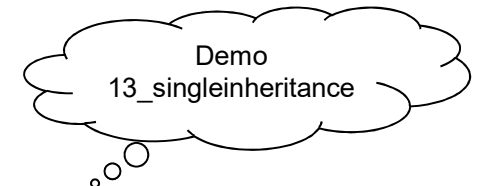
```
e = Employee("Employee 1")
w = WageEmployee("WageEmployee 1", 10, 10)

print(e.getName())
print(e.calcSalary())

print(w.getName())
print(w.calcSalary())
print(w.getWage())
print(w.getHours())
```

- **Result:**

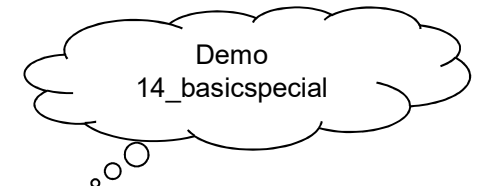
```
Employee 1
0
WageEmployee 1
0
10
10
```



# Inheritance

- **Class Special that inherits from a super-class Basic:**

```
class Basic:
 def __init__(self, name): self.name = name
 def show(self): print('Basic -- name: %s' % self.name)
class Special(Basic):
 def __init__(self, name, edible):
 Basic.__init__(self, name)
 self.upper = name.upper()
 self.edible = edible
 def show(self):
 Basic.show(self)
 print('Special -- upper name: %s.' % self.upper)
 if self.edible:
 print("It's edible.")
 else:
 print("It's not edible.")
 def edible(self):
 return self.edible
```





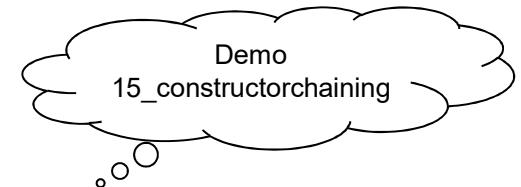
# Constructor Chaining

```
class A(object):
 def __init__(self):
 print("Constructor A was called")

class B(A):
 def __init__(self):
 super().__init__()
 print("Constructor B was called")

class C(B):
 def __init__(self):
 super().__init__()
 print("Constructor C was called")

c = C()
```



# Multiple Inheritance

- **Class can be derived from multiple parent classes:**

```
class A: # define your class A
.....
class B: # define your class B
.....
class C(A, B): # subclass of A and B
```

- **Functions to check relationships of two classes and instances:**

- `issubclass()` or `isinstance()`

- `issubclass(sub, sup)` **boolean function:**

- Returns true if the given subclass sub is a subclass of the superclass sup

- `isinstance(obj, Class)` **boolean function:**

- Returns true if obj is instance of class Class or is an instance of a subclass of Class

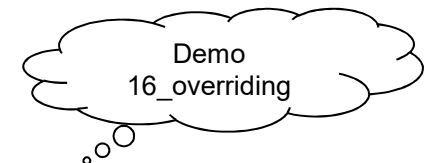
# Overriding Methods

- **Always possible to override parent class methods:**
  - Reason could be that special or different functionality in your subclass is needed

```
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
```

- **Result:**  
Calling child method



# Override Base Methods

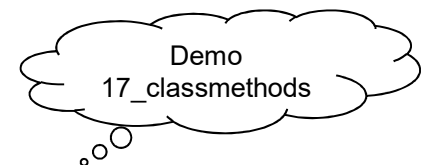
- **Generic functionality that you can override in your own classes:**
- `__init__ (self [,args...]):`
  - Constructor (with any optional arguments)  
`obj = className (args)`
- `__del__ (self):`
  - Destructor, called by system when object is actually deleted  
`del obj`
- `__repr__ (self):`
  - Evaluatable string representation  
`repr (obj)`
- `__str__ (self):`
  - Printable string representation  
`str (obj)`

# Class Methods

- **Class methods receive class as first parameter:**
  - Aren't specific to any particular instance, but still involve the class in some way
  - Class methods can be overridden or redefined by subclasses
- **Method can be made a class method in two ways:**
  - Annotate method with `@classmethod` decoration

```
class Bank:
 @classmethod
 def is_valid(cls, key): # code for determining validity here
 return valid
 def add_key(self, key, val):
 if not Bank.is_valid(key):
 raise ValueError()
Use method without instance, signals code closely-associated with Bank
Bank.is_valid('my key')
```

- `class` **method typically is called through the class**

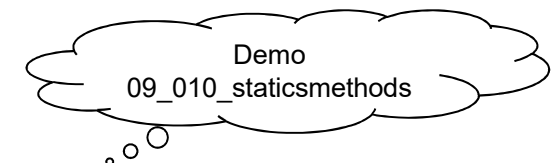


# Static Methods

- **Static methods do not receive** `self` or `cls` **as first parameter :**
  - Like regular function but has to be called in namespace of class
- **Method can be made** `static` **in two ways :**
  - Annotate method with `@staticmethod` decoration

```
class Account :
 interestRate = 10
 def getInterestRate1():
 return Account.interestRate
 getInterestRate1 = staticmethod(getInterestRate1)
 @staticmethod
 def getInterestRate2():
 return Account.interestRate

print (Account.getInterestRate1())
print (Account.getInterestRate2())
```



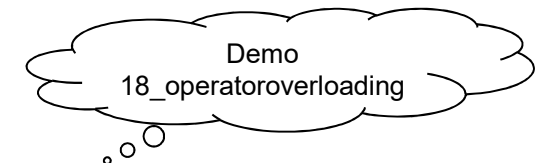
# Operator Overloading

- **Suppose** `Vector` **class represents two-dimensional vectors:**
  - Plus operator will not work to add them automatically
- **Define** `__add__` **method to perform vector addition:**
  - Plus operator would behave as per expectation

```
class Vector:
 def __init__(self, a, b):
 self.a = a
 self.b = b
 def __str__(self):
 return 'Vector (%d, %d)' % (self.a, self.b)
 def __add__(self, other):
 return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print(v1 + v2)
```

- **Result:** `Vector(7,8)`



# Polymorphism

- Call of same function / method on different types of objects is handled by different forms of that function
- Generic method call results in specific runtime action

```
cir = Circle(1, 2)
rec = Rectangle(3, 4, 2, 2)
list = [cir, rec]

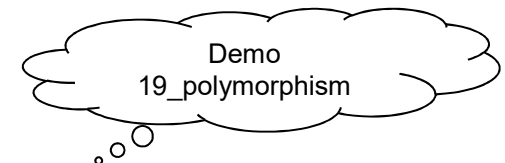
for item in list:
 item.draw();
```

In static-type languages only works if those objects derive from same super-class or interface. In Python you can arrange it that way, but no need to: **duck typing** implies that it is acceptable (works) if you just try it (with proper handling of cases when this fails)



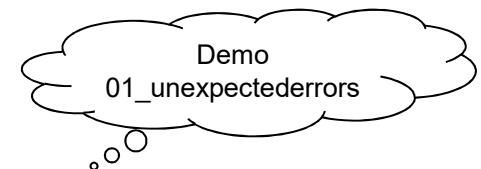
# Polymorphism in Python

```
class A(object):
 def show(self, msg):
 print('class A -- msg: "%s"' % (msg,))
class B(object):
 def show(self, msg):
 print('class B -- msg: "%s"' % (msg,))
class C(object):
 def show(self, msg):
 print('class C -- msg: "%s"' % (msg,))
def test():
 objs = [A(), B(), C(), A(),]
 for idx, obj in enumerate(objs):
 msg = 'message # %d' % (idx + 1,)
 obj.show(msg)
if __name__ == '__main__':
 test()
```



# Unexpected Errors

- **Python provides exceptions to handle unexpected errors:**
  - Exceptions are events disrupting normal program flow
  - May occur during the execution of program instructions
- **When Python encounters conditions it can't cope with:**
  - An exception is raised
  - An exception is a Python object that represents an error
- **When a Python script raises an exception:**
  - The exception must be handled immediately otherwise the script terminates
- **Another feature to handle unexpected errors are assertions:**
  - Assertions are tests which must be true to continue

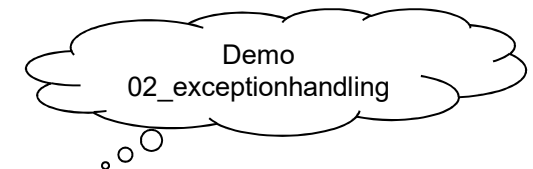


# Exception Handling

- **Place code that might raise an exception in a `try` block:**
  - Code in a `try` block is tested for exceptions
- **Include an `except` statement after the `try` block:**
  - Place code to handle the problem as elegantly as possible there
  - Generic `except` clause handles any exception

```
try:
 x = 1
 y = 0
 z = x/y # ZeroDivisionError
except:
 print('Cannot divide by zero')
try:
 fh = open("somefile", "r") # IOError
except:
 print('Cannot open file')
```

- **After handling exception in `except` statement:**
  - Normal program flow continues

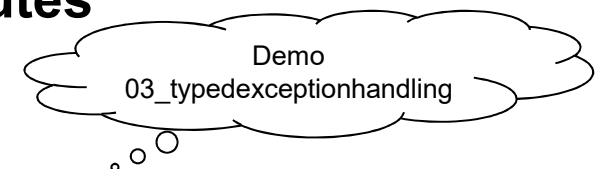


# Typed Exception Handling

- **except clauses can be conditioned on a type:**
  - Single `try` statement can have multiple typed `except` statements (first match executed)
  - Useful when the `try` block might throw different types of exceptions

```
try:
 x = 1
 y = 0
 z = x/y # ZeroDivisionError
 tup = (1,2,3)
 tup[0] = 2 # TypeError
except ZeroDivisionError:
 print('Division by zero')
except TypeError:
 print('Cannot change tuple contents')
```

- **except corresponding to exception type executes**



# Exception Handling with Else

- **After** `except` **clause(s)** `else` **clause can be included:**
  - Executes if the code in the `try:` block does not raise an exception

```
try:
```

```
 Do you operations here
```

```
except:
```

```
 If there is any exception, then execute this block
```

```
else:
```

```
 If there is no exception then execute this block
```

- **Example:**

```
try:
```

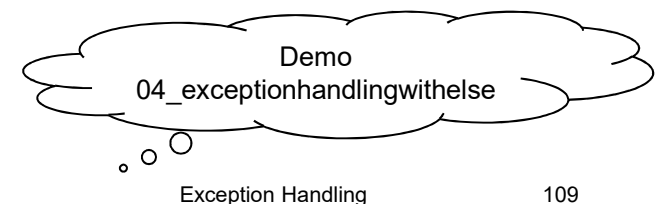
```
 x = int(input("Please enter a number: "))
```

```
except ValueError:
```

```
 print("Oops! That was not valid number")
```

```
else:
```

```
 print("Yeah! That was a really good number")
```

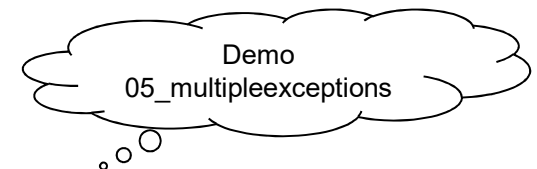


## except Clause Multiple Exceptions

- **Same** `except` **clause** may also handle multiple exceptions:

```
try:
 Do you operations here;
except(Exception1[, Exception2[,...ExceptionN]]):
 If there is any exception from the given exception list,
 then execute this block
else:
 If there is no exception then execute this block
```

- **Disadvantage** is that **error messages** are **less distinctive**

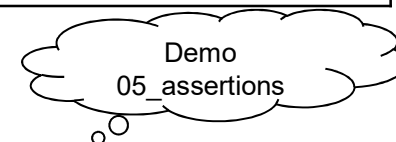


# Python Standard Exceptions

| Name               | Description                                                                                      |
|--------------------|--------------------------------------------------------------------------------------------------|
| Exception          | Base class for all exceptions                                                                    |
| StopIteration      | Raised when <code>next()</code> of iterator does not point to any object                         |
| SystemExit         | Raised by the <code>sys.exit()</code> function                                                   |
| StandardError      | Base class for built-in exceptions except <code>StopIteration</code> and <code>SystemExit</code> |
| AritmeticError     | Base class for all errors that occur for numeric calculation                                     |
| OverflowError      | When calculation exceeds maximum limit for numeric type                                          |
| FloatingPointError | Raised when a floating point calculation fails                                                   |
| ZeroDivisionError  | Raised when division by zero occurs for numeric types                                            |
| AssertionError     | Raised in case of failure of the <code>Assert</code> statement                                   |
| AttributeError     | Raised in case of failure of attribute reference or assignment                                   |
| EOFError           | Raised when end of file is reached                                                               |
| ImportError        | Raised when an import statement fails                                                            |
| TypeError          | Raised when operation attempted that is invalid for data type                                    |
| ValueError         | Raised when function for data type has invalid values                                            |
| RuntimeError       | Raised when generated error does not fall into any category                                      |

# Python Standard Exceptions

| Name                           | Description                                                                                                                                     |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>KeyboardInterrupt</code> | Raised when user interrupts program execution, usually by pressing CTRL-C                                                                       |
| <code>LookupError</code>       | Base class for all lookup errors                                                                                                                |
| <code>IndexError</code>        | Raised when an index is not found in a sequence                                                                                                 |
| <code>KeyError</code>          | Raised when the specified key is not found in the dictionary                                                                                    |
| <code>NameError</code>         | Raised when an identifier is not found in the local or global namespace                                                                         |
| <code>UnboundLocalError</code> | Raised when trying to access a local variable in a function or method but no value has been assigned to it                                      |
| <code>EnvironmentError</code>  | Base class for all exceptions that occur outside the Python environment                                                                         |
| <code>IOError</code>           | Raised when an input /output operation fails, such as the print statement or the open() function when trying to open a file that does not exist |
| <code>OSError</code>           | Raised for operating system related errors                                                                                                      |
| <code>SyntaxError</code>       | Raised when there is an error in Python syntax                                                                                                  |
| <code>IndentationError</code>  | Raised when indentation is not specified properly                                                                                               |
| <code>SystemError</code>       | Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit                  |





## try-finally Clause

- `finally` **block may be used along with a `try` block:**
  - `finally` block is always executed
  - Whether or not the try-block raised an exception

- **Syntax of the `try-finally`:**

```
try:
```

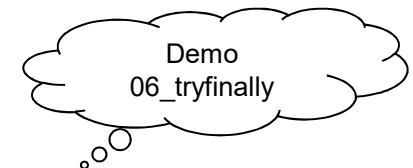
```
 Do you operations here;
```

```
 Due to any exception, this may be skipped
```

```
finally:
```

```
 This would always be executed
```

- **Combining `finally` and `except` is allowed:**
  - `try` block can be followed by multiple `except` clauses and one `finally` clause



# Exception Arguments

- **An exception can have an argument:**
  - Value that gives additional information about the problem
  - Contents of the argument vary by exception
- **Capture argument by supplying a variable in `except` clause:**

```
try:
 Do you operations here;
except ExceptionType as var:
 You can print value of var here...
```

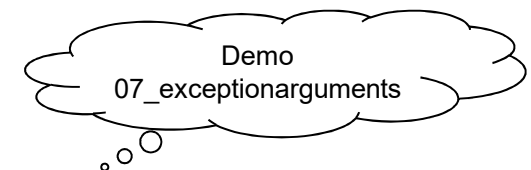
- **Variable will receive the value of the exception:**
  - Can receive a single value or multiple values in the form of a tuple
  - Tuple usually contains the error string, the error number and an error location

## Example Argument of an Exception

```
Define a function here.
def temp_convert(var):
 try:
 return int(var)
 except ValueError as e:
 print("Argument does not contain numbers\n", e)
Call above function here
temp_convert("xyz");
```

- **Result:**

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```



# Raising Exceptions

- **Exceptions can be raised with the `raise` statement:**

```
raise # re-raise current exception in except statement
raise Exception(args) # raise new anywhere
raise Exception(args) from e # include current in new
```

- **Explanation:**

- Exception is the type of exception, for example `NameError`
- Argument is a value for the exception argument
- The argument is optional; if not supplied, the exception argument is `None`.
- Final argument `traceback` is optional and is the `traceback` object for exception

- **Example:**

- An exception can be a string, a class, or an object
- Most of the exceptions that the Python core raises are classes  
— with an argument that is an instance of the class

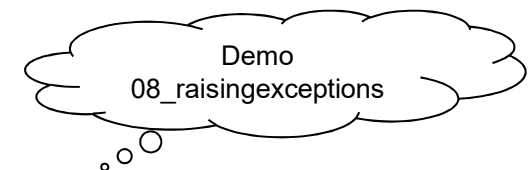
## Example Raising an Exception

```
try:
 raise NameError('HiThere')
```

In order to catch an exception:

- `except` should refer to same exception thrown as class object
- **To capture above exception, write except clause as follows:**

```
try:
 Business Logic here...
except NameError:
 print('An exception flew by!')
 # raise
else:
 Rest of the code here...
```



# User Defined Exceptions

- **Created by deriving classes from standard built-in exceptions:**
  - Exception class may created that is subclassed from `RuntimeError`
  - Useful when more specific information is needed when an exception is caught

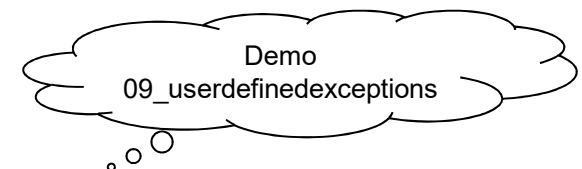
- **In `try: block`:**

- User-defined exception is raised and caught in the `except` block
- The variable `e` is used to create an instance of the class `Networkerror`

```
class Networkerror(RuntimeError):
 def __init__(self, arg):
 self.args = arg
```

- **With above class exception is raised as follows:**

```
try:
 raise Networkerror("Bad hostname")
except Networkerror as e:
 print(e.args)
```



# Input and Output

- **Simplest way to produce output is `print` function:**

- Allows you to pass expressions, separated by commas, enclosed in parentheses

```
print("Python is really a great language,", "isn't it? ")
```

- **Major difference between version 2 and 3 of the language:**

- In version 3 `print` is a function

- **Standard input default comes from keyboard with `input` function**

- `input([prompt])` **function:**

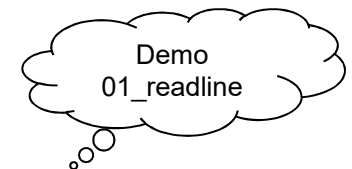
- Reads line from standard input, converts it to a string and strips trailing newline

- **Python also has File Object methods:**

- Most of the file manipulation is done using a file object

- **Python also has OS Object Methods:**

- Methods to process files as well as directories



# IO Module

- **Core of the I/O system is implemented in io library module**
- **Consists of a collection of different I/O classes:**
  - `FileIO`
  - `BufferedReader`
  - `BufferedWriter`
  - `BufferedRWPair`
  - `BufferedRandom`
  - `TextIOWrapper`
  - `BytesIO`
  - `StringIO`
- **Each class implements a different kind of I/O**
- **Classes get layered to add features**



# Opening Files

- **open function:**

- Opens a file to be able to read or write a file later
- Creates file object which can be used to call other support methods

```
file_object = open(file ,mode=r, buffering=-1, encoding=None)
```

- **file:**

- String value that contains the name of the file to be accessed

- **mode:**

- Determines mode in which the file has to be opened, read, write, append etc.
- `t` for text mode is default and optional, `b` is for byte mode

- **buffering:**

- If buffering value is set to 0, no buffering will take place
- If buffering value is 1, line buffering will be used (in text mode)
- If buffering > 1: buffering performed with indicated buffer size
- Default (-1): fixed-sized buffering (heuristic or `io.DEFAULT_BUFFER_SIZE`; line if tty)

# File Open Modes

- **Python distinguishes between binary and text I/O:**
  - In binary mode content is returned as bytes objects without any decoding
  - In text mode as `str`, bytes decoded using given or platform-specific encoding

| Modes            | Description                                                                                                                                                                                                  |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>r</code>   | Opens file for reading only. File pointer is placed at beginning of the file. Default mode                                                                                                                   |
| <code>rb</code>  | Opens file for reading only in binary format. File pointer is placed at beginning of file                                                                                                                    |
| <code>r+</code>  | Opens file for both reading and writing. File pointer will be at the beginning of the file                                                                                                                   |
| <code>rb+</code> | Opens file for both reading and writing in binary format. File pointer will be at the beginning of the file                                                                                                  |
| <code>w</code>   | Opens file for writing only. Overwrites file if file exists. If file does not exist, creates a new file for writing                                                                                          |
| <code>wb</code>  | Opens file for writing only in binary format. Overwrites file if file exists. If file does not exist, creates new file for reading and writing                                                               |
| <code>w+</code>  | Opens file for writing and reading. Overwrites existing file if file exists. If file does not exist, creates new file for reading and writing                                                                |
| <code>wb+</code> | Opens file for both writing and reading in binary format. Overwrites existing file if file exists. If the file does not exist, creates a new file for reading and writing                                    |
| <code>a</code>   | Opens a file for appending. File pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.               |
| <code>ab</code>  | Opens a file for appending in binary format. File pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates new file for writing |

# File Object Attributes

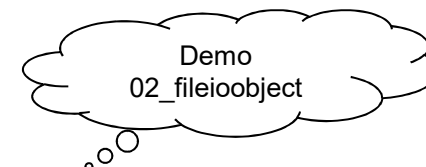
- **FileIO object represents raw unbuffered binary I/O**
- **File object has attributes with information about it**
- `file.closed` : **Returns true if file is closed, false otherwise**
- `file.mode` : **Returns access mode with which file was opened**
- `file.name` : **Returns name of the file**

- **Example:**

```
fo = open("test.txt", "wb")
print("Name of the file: ", fo.name)
print("Closed: ", fo.closed)
print("Opening mode: ", fo.mode)
```

- **Result:**

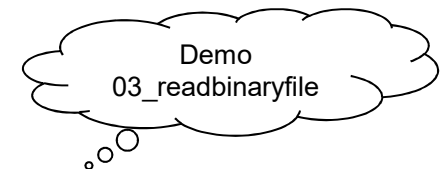
```
Name of the file: test.txt
Closed or not: False
Opening mode: wb
```



# Reading Binary Files

- **Opening a file in binary mode is simple but subtle:**
  - Reading bytes not strings, so there's no conversion for Python to do
- **Difference from opening it in text mode:**
  - b mode parameter
  - No encoding attribute
- **read method reads whole file or a number of bytes:**
  - `tell` moves file pointer to the beginning of the file
  - `seek` moves file pointer to a certain position in the file

```
fr = open('trilobyte.jpg', mode='rb')
fr.tell()
data = fr.read(3) => read three bytes
print(type(data)) => type is bytes
fr.tell()
fr.seek(0)
data = fr.read()
print(data)
```



# Writing Binary Files

- **Open binary files for writing without encoding attribute:**

```
fw = open('binaryfile04.txt', mode='wb')
```

- **Use `pack` method of `struct` module to prepare buffer for writing:**
  - Format string determines how packing takes place

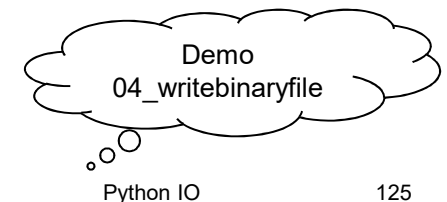
```
nr1 = 1; nr2 = 2; result = nr1 + nr2
buffer = pack("iii", nr1, nr2, result)
fw.write(buffer)
```

- **Data must be read like it is written using same format string:**

```
fr = open('binaryfile04.txt', mode='rb')
buffer = fr.read()
data = unpack("iii", buffer)
```

- **Returned data is a tuple that can be accessed as follows:**

```
num1 = data[0], num2 = data[1], res = data[2]
```



## Writing Text Files

- **Open file for writing specifying mode and encoding:**

```
fw = open('textfilepolish.txt', mode='w', encoding='UTF-8')
```

- **write method writes any string to an open file:**

- Parameter is the content to be written into the opened file
- Does not add a newline character ('\n') to the end of the string

```
s = 'Polish text: aćęłńóśźżĄĆĘŁŃÓŚŹŻ'
fw.write(s)
```

- **Always close the file when you are done:**

```
fw.close()
```

- **File textfilepolish.txt with following content is created:**

```
Polish text: Ä...Ä†Ä™Ä, Ä,,Ä³Ä>Ä°Ä¼Ä,,Ä†ÄÄÄfÄ"ÄšÄ¹Ä»
```

# Reading Text Files

- **Open file for reading specifying mode and encoding:**

```
fr = open('textfilepolish.txt', mode='r', encoding='UTF-8')
```

- **read method reads a string from an open file:**

- Parameter is the number of bytes to be read from the opened file

```
fileObject.read([count]);
```

- **read() starts reading from the beginning of the file:**

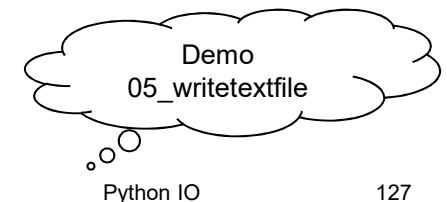
- If count is missing then it tries to read as much as possible, may be until end of file

- **Example of reading and closing file** `textfilepolish.txt`:

```
data = fr.read()
print(data)
fr.close()
```

- **Result:**

Polish text: `ąćęłńóśźżĄĆĘŁŃÓŚŹŻ`



# Closing Files

- **`close` method flushes unwritten data and closes file object:**
  - No more writing can be done after calling `close`
  - File is closed automatically when reference file object is reassigned to another file

- **Syntax:**

```
fileObject.close();
```

- **Example:**

```
fo = open("foo.txt", "wb")
print("Name of the file: ", fo.name)
fo.close()
```

- **Result:**

```
Name of the file: foo.txt
```

- **Good practice to use `close()` method to close a file**



# File Positions

- `tell` **method retrieves current position within the file:**
  - Next read or write will occur at that many bytes from the beginning of the file
- `seek(offset[, from])` **method changes the current file position:**
  - Offset argument indicates the number of bytes to be moved
  - From argument specifies reference position from where bytes are to be moved
- **If from is set to 0:**
  - Use beginning of the file as the reference position
- **If from is set to 1:**
  - Use current position as the reference position
- **If from is set to 2:**
  - Use end of the file as the reference position

## Example File Positioning

- **Consider file** `foo.txt` **that was created before:**

```
fo = open("foo.txt", "r+")
str = fo.read(10);
print("Read String is: ", str)
Check current position
position = fo.tell();
print("Current file position: ", position)
Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10);
print("Again read String is: ", str)
fo.close()
```

- **Result:**

```
Read String is: Python is
Current file position: 10
Again read String is: Python is
```

# Renaming and Deleting Files

- **Files can be renamed and deleted:**
  - `os` module has methods for file-processing operations like renaming and deleting
  - `os` module must first be imported before methods can be called

- `rename` **method takes two arguments:**

- Current filename and new filename

```
os.rename(current_file_name, new_file_name)
```

- **Rename an existing file from `test1.txt` to `test2.txt`:**

```
import os
os.rename("test1.txt", "test2.txt")
```

- `delete` **method deletes files supplied as argument:**

```
os.delete(file_name)
```

- **Delete an existing file `test2.txt`:**

```
import os
os.remove("text2.txt")
```

# Creating and Deleting Directories

- **Files are in directories and Python can handle directories too:**

- os module has methods to create, remove, and change directories

- `mkdir()` **method creates directory in the current directory:**

- Argument is name of the directory to be created

- **Syntax:** `os.mkdir("newdir")`

- **Example to create a directory test in current directory:**

```
import os
os.mkdir("test")
```

- `rmdir()` **method deletes the directory passed as an argument:**

- Before removing a directory, all the contents in it should be removed

- **Syntax:** `os.rmdir('dirname')`

- **Example to remove `"/tmp/test"` directory:**

- Give fully qualified name of directory otherwise search directory in current directory

```
import os
os.rmdir("/tmp/test")
```

## Directory Methods

- `chdir()` **changes the current directory:**
  - Argument is the name of the directory that you want to make the current directory

- **Syntax:**

```
os.chdir("newdir")
```

- **Example changing a directory to `"/home/newdir"`:**

```
import os
os.chdir("/home/newdir")
```

- `getcwd()` **displays the current working directory**

- **Syntax:**

```
os.getcwd()
```

- **Example to give location of the current directory:**

```
import os
os.getcwd()
```

# Iteration

- **Python `for` statement can iterate over many kinds of objects**
- **Iterate over a collection of items:**

```
for x in [1,4,5,10]:
 print(x) # 1 4 5 10
```

- **Iteration over dictionary given the keys:**

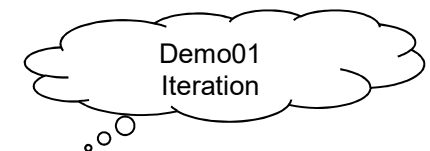
```
prices = { 'GOOGLE': 490.10, ' IBM': 145.23, 'YAHOO': 21.71 }
for key in prices:
 print(key) # GOOGLE IBM YAHOO
```

- **Iteration over a string gives characters in string :**

```
s = "Mars!"
for c in s:
 print(c) # M a r s !
```

- **Iteration over a file gives the lines:**

```
for line in open("realprogrammers.txt"):
 print(line)
```



# Consuming Iterables

- **Many functions can consume an iterable object:**

- **Reductions:**

```
s = [1,4,5,10]
sum(s)
min(s)
max(s)
```

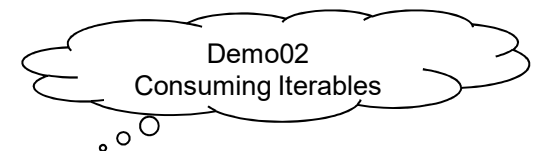
- **Constructors:**

```
l = list(s)
t = tuple(s)
st = set(s)
```

- **in operator:**

```
item in s
```

- **And many others in the library**



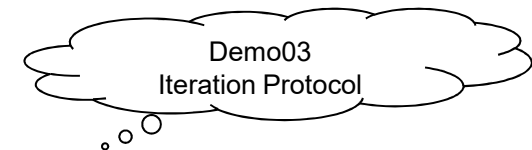
# Iteration Protocol

- **Protocol implemented by objects that you can iterate over:**

```
items = [1, 4, 5]
it = items.__iter__()
print(next(it))
print(next(it))
print(next(it))
print(next(it))
Traceback (most recent call last):
File "D:\..\src\demo03_iteration_protocol.py", line 7, in <module>
 print(next(it))
StopIteration
```

- **Underneath the covers of for statement:** `for x in obj:`

```
_iter = obj.__iter__() # Get iterator object
while True:
 try:
 x = next(_iter) # Get next item
 except StopIteration: # No more items
 break
 statements
```



- **Object supporting `__iter__()` and `__next__()` is iterable**



# Supporting Iteration

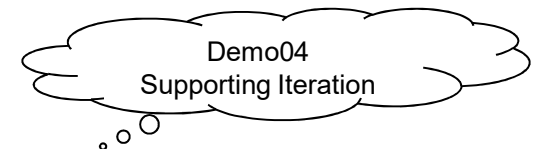
- **User-defined objects can support iteration:**

- Object implement should implement `__iter__()` and `__next__()`

```
class Countdown(object):
 def __init__(self, start):
 self.count = start
 def __iter__(self):
 return self
 def __next__(self):
 if self.count <= 0:
 raise StopIteration
 r = self.count
 self.count -= 1
 return r
```

- **Now Countdown can be iterated as follows:**

```
c = Countdown(5)
for i in c:
 print(i)
```



# Generators

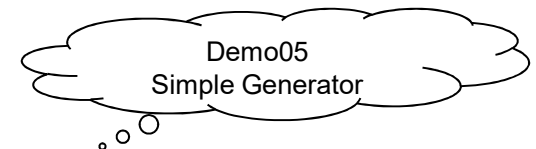
- **Functions producing sequence of results instead of single value:**

```
def countdown(n):
 while n > 0:
 yield n
 n -= 1
for i in countdown(5):
 print(i)
```

- **Instead of returning a value:**
  - Generates series of values using `yield` statement
- **Behavior is quite different from normal function:**
  - Calling a generator function creates an generator object
  - However, it does not start running the function

```
x = countdown(10)
print(x) # <generator object at 0x58490>
```

- **Notice that no output was produced**



# Generator Functions

- **Function only executes on** `next()` :


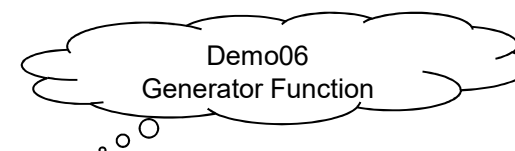
```
x = countdown(6)
print(x) # <generator object countdown at 0x00D5B4E0>
```

- **yield produces a value, but suspends the function:**
  - Function resumes on next call to `next()`

- **Counting down from 10:**

```
print(next(x)) #6
print(next(x)) #5
print(next(x)) #4
print(next(x)) #3
print(next(x)) #2
print(next(x)) #1
print(next(x)) # Exception => Stopiteration
```

Function starts  
executing here

## Convenient Iterator

- **Generator function mainly more convenient to write an iterator:**
  - Don't worry about iterator protocol with `__next__()` and `__iter__()`
  - It just works
- **Generators versus iterators:**
  - Generator function is slightly different than an object that supports iteration
  - Generator is a one-time operation:
    - Can iterate over the generated data once
    - If you want to do it again, you have to call the generator function again
  - This is different than a list:
    - You can iterate over a list as many times as you want

# Generator Expression

- **Generated version of a list comprehension:**

```
li = [1,2,3,4]
generator expression, watch the parenthesis
ge = (2*x for x in li)
print(ge) # <generator object <genexpr> at 0x0085B4E0>
print(next(ge)) # prints 2
for i in ge: # prints 4, 6 and 8
 print(i)
```

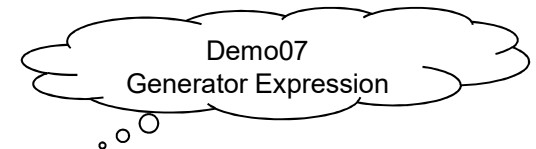
- **Loops over sequence of items and applies operation to each:**

- However, results are produced one at a time using a generator

- **Important differences from a list comprehension:**

- Generator Expression does not construct a list
- Its only useful purpose is iteration and once consumed it can't be reused

```
list comprehension, watch the block brackets
lc = [2*x for x in li]
print(lc) # [2, 4, 6, 8]
```



# Generator Expression Syntax

- **General syntax:**

```
(expression for i in s if cond1
 for j in t if cond2
 ...
 if condfinal)
```


- **Resolves to:**

```
for i in s:
 if cond1:
 for j in t:
 if cond2:
 ...
 if condfinal: yield expression
```

- **If used as a single function argument:**

- Parenthesis on a generator expression can be dropped

```
sum(x*x for x in s)
```

  
 Generator expression

# Building Blocks

- **Generator functions:**

```
def countdown(n):
 while n > 0:
 yield n
 n -= 1
```

- **Generator expressions:**

```
squares = (x*x for x in s)
```

- **In both cases:**

- Get object that generates values which are typically consumed in a for loop

# Programming Problem

- **Find out how many bytes of data were transferred:**

- Sum up last column of data in huge Apache web server log

```
81.107.39.38 - ... "GET /ply/ HTTP/1.1" 200 7587
81.107.39.38 - ... "GET /favicon.ico HTTP/1.1" 404 133
81.107.39.38 - ... "GET /ply/bookplug.gif HTTP/1.1" 200 23903
81.107.39.38 - ... "GET /ply/ply.html HTTP/1.1" 200 97238
81.107.39.38 - ... "GET /ply/example.html HTTP/1.1" 200 2359
66.249.72.134 - ... "GET /index.html HTTP/1.1" 200 4447
```

- **Each line of log looks like this:**

```
81.107.39.38 - ... "GET /ply/ply.html HTTP/1.1" 200 97238
```

- **Number of bytes is the last column:**

```
bytestr = line.rsplit(None,1)[1]
```

- **Is either a number or a missing value (-):**

```
81.107.39.38 - ... "GET /ply/ HTTP/1.1" 304 -
```



## Problem Solutions

- **Non-Generator Solution:**

- Do a simple for-loop, read-by-line and update a sum:

```
wwwlog = open("access-log")
total = 0
for line in wwwlog:
 bytestr = line.rsplit(None,1)[1]
 if bytestr != '-':
 total += int(bytestr)
print("Total", total)
```

- **Generator Solution:**

- Use some generator expressions

```
wwwlog = open("access-log")
bytecolum = (line.rsplit(None,1)[1] for line in wwwlog)
bytes = (int(x) for x in bytecolum if x != '-')
print("Total", sum(bytes))
```

- **Notice lesser code and a completely different programming style**

# Functional Programming

- **Programming style that is focused on expressions:**
  - Also called expression oriented programming
- **Python has functions and features enabling functional approach:**
  - `map, filter, reduce, lambda`
  - `list comprehension`
- `map(aFunction, aSequence):`
  - Applies operation to each item and collects the result
- `filter(aFunction, aSequence):`
  - Extracts each element in the sequence for which function returns True
- `reduce(aFunction, aSequence):`
  - Reduces list to single value by combining elements via supplied function
  - In `functools` in Python 3.0
- `lambda :`
  - Small anonymous function consisting of single line

# Map and Filter

- **Higher-order functions** `map` and `filter` that operate on lists:

```
def square(x):
 return x*x
def even(x):
 return 0 == x % 2

some_list = list(map(square, range(10,20)))
print(some_list)
```

- **Output:** [100, 121, 144, 169, 196, 225, 256, 289, 324, 361]

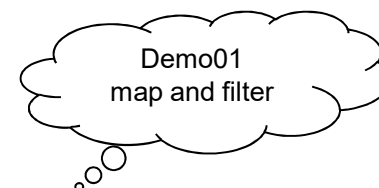
```
some_list = list(filter(even, range(10,20)))
print(some_list)
```

- **Output:** [10, 12, 14, 16, 18]

```
some_list = list(map(square, filter(even, range(10,20))))
print(some_list)
```

- **Output:** [100, 144, 196, 256, 324]

- **More Pythonic are list comprehensions**



## Reduce and Lambda

- `reduce` **accepts iterator to process and returns single result:**

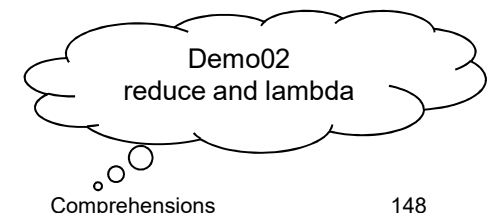
```
from functools import reduce
var1 = reduce((lambda x, y: x * y), [1, 2, 3, 4])
print(var1) # 24
```

- **Above code works as follows:**

- `reduce` passes current product along with next item from list to `lambda` function
- Default first item in sequence initialized starting value

- **Same functionality with `for` loop:**

```
li1 = [1, 2, 3, 4]
result = li1[0]
for x in li1[1:]:
 result = result * x
print(result)
```



# List Comprehensions

- **Construct for creating new list based on existing lists:**
  - Languages like Haskell, Erlang, Scala and Python have them
- **Comprehension term:**
  - Comes from math's set comprehension notation to define sets in terms of other sets
- **Powerful and popular feature in Python:**
  - Generate new list by applying operation or function to every member of original list

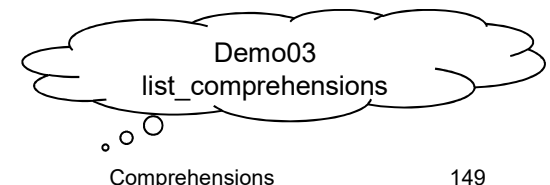
```
slist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(slist)
```

```
list_str = [str(x) for x in slist]
print(list_str)
```

- **Output:** ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']

```
new_list = [x * 2 for x in slist]
print(new_list)
```

- **Output:** [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]



# Syntax List Comprehensions

- **Syntax of a list comprehension may involve:**
  - `for` loop, an `in` operation and an `if` statement
- **Python's notation:**
  - `[ expression for name in list ]`
  - Expression:
    - Some calculation or operation acting upon the variable name
  - For each member of the list, the list comprehension
    - Sets name equal to that member
    - Calculates a new value using expression
  - Collects these new values into a list which is return value of list comprehension

```
li = [3, 6, 2, 7]
li_new = [elem*2 for elem in li] => [6, 12, 4, 14]
```
- **`if` statement acts as a filter and is optional:**

```
[x-10 for x in grades if x>0]
```

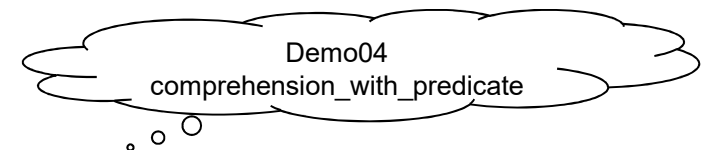
# Filtered List Comprehension

- **Filter determines if expression is performed on list member:**
  - For each element of list, checks if it satisfies the filter condition
- **If filter condition returns False:**
  - Element is omitted from the list before the list comprehension is evaluated
- **Traditional syntax:**

```
slist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for x in slist:
 if (x % 2) == 0:
 new_list.append(str(x))
print(new_list) => ['2', '4', '6', '8', '10']
```

- **Pythonic syntax using list comprehension:**

```
new_list = [str(x) for x in slist if (x % 2) == 0]
print(new_list) => ['2', '4', '6', '8', '10']
```



# Syntactic Sugar

- **List comprehensions are syntactic sugar for higher-order functions:**

```
[expression for name in list]
map(lambda name: expression, list)
```

- **Example using higher order function:**

```
lm = list(map(lambda x: 2*x+1, [10, 20, 30]))
print(lm) # [21, 41, 61]
```

- **Example using list comprehension:**

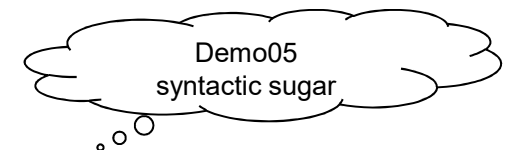
```
lc = [2*x+1 for x in [10, 20, 30]]
print(lc) # [21, 41, 61]
```

- **Example using higher order function:**

```
ls1 = list(map(lambda x: 2*x+1, filter(lambda x: x > 0, [10, 20, 30, -10])))
print(ls1)
```

- **Example using list comprehension:**

```
ls2 = [2*x+1 for x in [10, 20, 30, -10] if x > 0]
print(ls2)
```





# Nested List Comprehensions

- **List comprehensions can easily be nested:**
  - Take list as input and produce list as output
- **In following inner comprehension produces:** [4, 3, 5, 2]

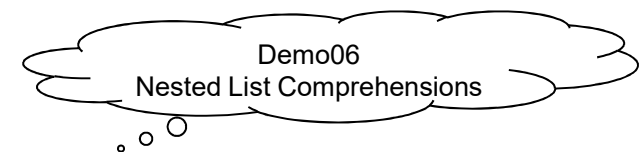
```
li = [3, 2, 4, 1]
[elem*2 for elem in [item+1 for item in li]] #[8, 6, 10, 4]
```

- **Another example is:**

```
list_a = ['A', 'B']
list_b = ['C', 'D']
list_c = [[x+y for x in list_a] for y in list_b]
print(list_c)
```

- **Produces:**

```
[['AC', 'BC'], ['AD', 'BD']]
```



# Comprehension Features

- **If list contains elements of different types:**
  - Expression must operate correctly on the types of all of list members.
- **If elements of list are other containers:**
  - Name can consist of container of names matching type and “shape” of list members

```
list_d = [('a' , 1), ('b', 2), ('c', 7)]
list_e = [n * 3 for (x, n) in list_d]
print(list_e) # [3, 6, 21]
```

- **Containers are objects that contain references to other objects:**
  - Lists, types, dictionaries
- **Expression can also contain user-defined functions:**

```
def subtract(a, b):
 return a - b
oplist = [(6, 3), (1, 7), (5, 5)]
list_f = [subtract(y, x) for (x, y) in opelist]
print(list_f) # ([-3, 6, 0])
```



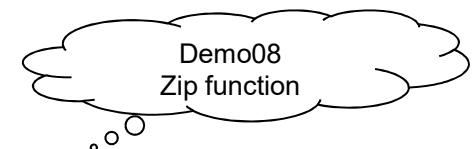
# Zip Function

- `zip(*iterables):`
  - Makes an iterator that aggregates elements from each of the iterables
  - iterator stops when the shortest input iterable is exhausted
- **Used to combine two lists in list of pairs:**

```
ls1 = [1,2,3]
ls2 = [4,5,6]
ls3 = list(zip(ls1,ls2))
print(ls3) # [(1, 4), (2, 5), (3, 6)]
```

- **Original lists can also be restored (as tuples) using same function:**

```
b1, b2 = zip(*ls3)
print(b1) -> (1,2,3)
print(b2) -> (4,5,6)
```



## Dictionary Construction with zip

- **zip can be used to generate dictionaries:**
  - When keys and values must be computed at runtime

- **Dictionaries are traditionally created with:**

```
dict1 = {'a':1, 'b':2, 'c':3}
print(dict1) # {'a': 1, 'c': 3, 'b': 2}
```

- **Or we can make it by assigning values to keys:**

```
dict2 = {}
dict2['a'] = 1; dict2['b'] = 2; dict2['c'] = 3
print(dict2) # {'a': 1, 'c': 3, 'b': 2}
```

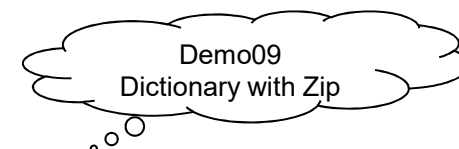
- **Sometimes keys and values are only know at runtime:**

- Can then use `zip` to create lists and loop through them in parallel like this:

```
dict3 = {}
for (k,v) in zip(keys, values):
 dict3[k] = v
print(dict3) # {'a': 1, 'b': 2, 'c': 3}
```

- **Alternative is using constructor with:**

```
dict4 = dict(zip(keys, values))
```



# Dictionary Comprehensions

- **Dictionary comprehension is like a list comprehension:**
  - Constructs a dictionary instead of a list
- **Dictionary comprehension makes dictionary from two lists:**

```
keys = ['a', 'b', 'c']
values = [1, 2, 3]

dict5 = { k:v for (k,v) in zip(keys, values)}
print(dict5) # {'a': 1, 'b': 2, 'c': 3}
```

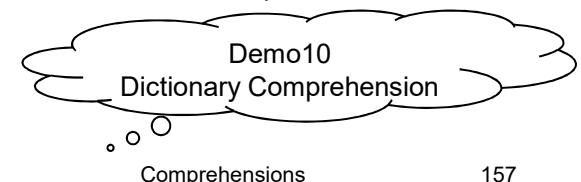
- **Requires actually more code then constructor function:**

```
dict6 = dict(zip(keys, values))
print(dict6) # {'b': 2, 'c': 3, 'a': 1}
```

- **Many more occasions to use dictionary comprehension:**

```
dict7 = {x: x**2 for x in [1,2,3,4,5]}
print(dict7) # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

dict8 = {x.upper(): x*3 for x in 'abcd'}
print(dict8) # {'A': 'aaa', 'C': 'ccc', 'B': 'bbb', 'D': 'ddd'}
```



## Dictionary from Keys

- **We can initialize dictionary from keys as follows:**

```
dict9 = dict.fromkeys(['a','b','c'], 0)
print(dict9) # {'a': 0, 'c': 0, 'b': 0}
```

- **Can use dictionary comprehension to do the same thing:**

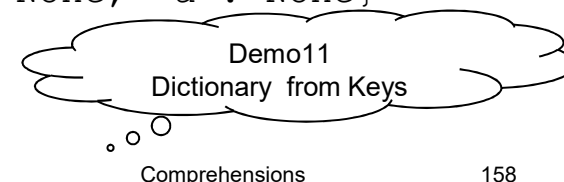
```
dict10 = {k: 0 for k in ['a','b','c']}
print(dict10) # {'a': 0, 'c': 0, 'b': 0}
```

- **Generate dictionary by iterating each element:**

```
dict11 = dict.fromkeys('dictionary')
print(dict11) # {'d': None, 'o': None, 'i': None,
't': None, 'a': None, 'c': None, 'y': None, 'n': None, 'r': None}
```

- **Now using comprehension:**

```
dict12 = {k:None for k in 'dictionary'}
print(dict12) # {'t': None, 'y': None, 'c': None,
'd': None, 'i': None, 'r': None, 'o': None, 'n': None, 'a': None}
```



# Set Comprehensions

- **Sets have their own comprehension syntax as well:**

- Quite similar to the syntax for dictionary comprehensions
- Difference is that sets just have values instead of `key:value` pairs

- **Set comprehensions can take set as input:**

- Following set comprehension calculates squares of set of numbers from 0 to 9

```
my_set = set(range(10))
print(my_set) # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
my_set2 = {x ** 2 for x in my_set}
print(my_set2) # {0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
```

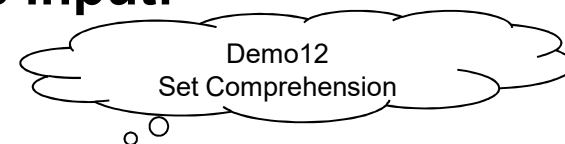
- **Set comprehensions can contain `if` clause to filter items:**

```
my_set3 = {x for x in my_set if x % 2 == 0}
print(my_set3) #{0, 8, 2, 4, 6}
```

- **Set comprehensions do not have to take set as input:**

- They can take any sequence

```
my_set4 = {2**x for x in range(10)}
print(my_set4) #{32, 1, 2, 4, 8, 64, 128, 256, 16, 512}
```



## Course Schedule

- Python Intro
- Variables and Data Types
- Data Structures
- Control Flow
- Functions
- Modules
- Classes and Objects
- Exception Handling
- Python IO
- Comprehensions
- Generators
- **Decorators**

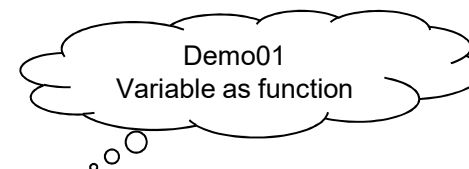
- Function as Objects
  - Passing and Returning Functions
  - What is a Decorator?
  - Decorator Syntax
  - Types of Decorators
  - Passing Arguments
  - Multiple Decorators
  - Class Decorators
  - Syntax Class Decorators
  - Singleton Class
  - Why Decorators
  - Need for AOP
  - Crosscutting Concerns
- 
- Python Database Access
  - Python and XML
  - Python Libraries



# Functions as Objects

- **Functions are first class citizens in Python**
- **Functions can be assigned to variables:**

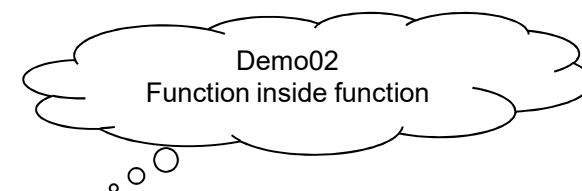
```
def greet(name):
 return "hello " + name
greet_someone = greet
print(greet_someone("Albert"))
=> Outputs: hello Albert
```



- **Function can be defined inside another function:**

```
def greet(name):
 def get_message():
 return "Hello "
 result = get_message()+name
 return result

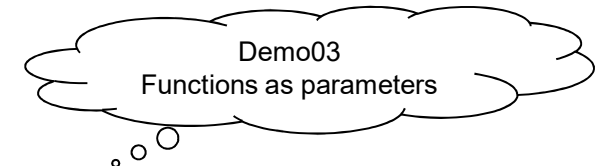
print(greet("Albert"))
=> Outputs: Hello Albert
```



# Passing and Returning Functions

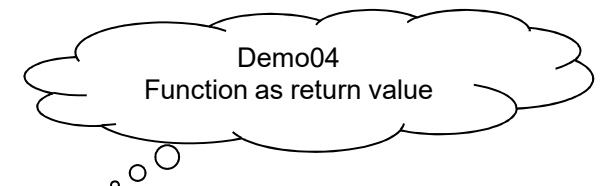
- **Python supports functional programming style**
- **Function can take other function as an argument:**

```
def greet(name):
 return "Hello " + name
def call_func(func):
 other_name = "Albert"
 return func(other_name)
print(call_func(greet)) => Outputs: Hello Albert
```



- **Function can return another function:**

```
def compose_greet_func():
 def get_message():
 return "Hello there!"
 return get_message
greet = compose_greet_func()
print(greet()) => Outputs: Hello there!
```



# Closure

- **Closure is combination of code and scope:**
  - Python functions combine code to be executed and scope in which to do that
- **Closure is mostly about nested function and scope of function:**
  - Function can retain value when it was created even though scope cease to exist

```
def startAt(start):
 def incrementBy(inc):
 return start + inc
 return incrementBy

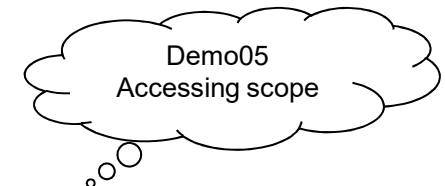
f = startAt(10)
g = startAt(100)

print f(1), g(2) # print 11 102
```

- **Inner function can access outer scope:**

```
def compose_greet_func(name):
 def get_message():
 return "Hello there "+name+"!"
 return get_message

greet = compose_greet_func("Albert")
print(greet()) # Outputs: Hello there Albert
```



# What is a Decorator?

- **Decorator is used to wrap a function:**
  - Gives new functionality without changing the original function
- **Following decorator wraps function output with bold tags:**

```
def get_text(name):
 return "Hello {0}, how are you".format(name)

def p_decorate(func):
 def func_wrapper(name):
 return "<p>{0}</p>".format(func(name))
 return func_wrapper

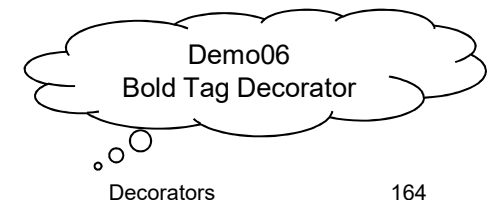
my_get_text = p_decorate(get_text)

print(my_get_text("Albert"))

=> Outputs <p>Hello Albert, how are you</p>

get_text = p_decorate(get_text)
print(get_text("Albert"))

=> Outputs <p>Hello Albert, how are you</p>
```



# Decorator Syntax

## Syntax:

```
def decorator_function():


```

```
@decorator_function
def my_func:


```

## This is equivalent to:

```
my_func =
decorator_function(my_func)
```

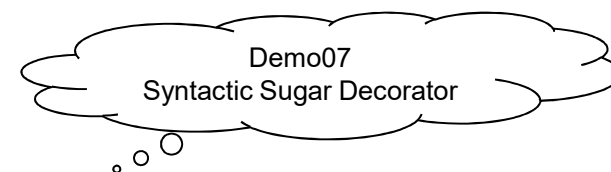
- **Decorator function:**

- Takes function being decorated as parameter
- Wraps it in a wrapper function and returns wrapper function

```
def decorator_function(decorated_function):
 def wrapper_function():

 some code -----
 decorated_function()

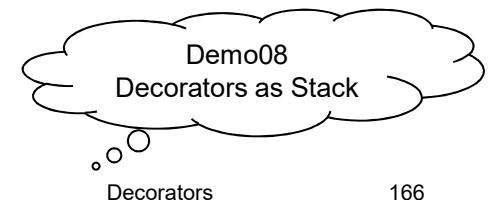
 return wrapper_function
```



# Types of Decorators

- **There are two types of decorators:**
  - Built-in decorators and User Defined decorators
- **Built in decorators are** `staticmethod` **and** `classmethod`:
  - Used in classes to define class or hierarchy specific methods
  - Also adressable with `@staticmethod` or `@classmethod` annotations
- **User defined decorators:**
  - Can be more that one decorator for one function and executed using stack
- **First all decorators are pushed into a stack:**
  - Then they are popped one by one

```
@helloGalaxy
@helloSolarSystem
def hello():
 print("Hello World")
hello()
```



# Passing Arguments

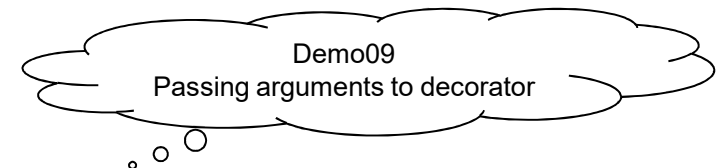
- **Wrapper function should accept arguments:**
  - If arguments which being passed to the function being decorated

```
def helloSolarSystem(old_function):
 def wrapper_function(planet=None):
 print("Hello Solar System")
 old_function(planet)
 print("Leaving Solar System")
 return wrapper_function

@helloSolarSystem
def hello(planet=None):
 if planet:
 print("Hello "+planet)
 else:
 print("Hello World")

hello("Mars")
hello()
```

- **Decoration can take place:**
  - Before, after or around the original function call



# Variable Arguments

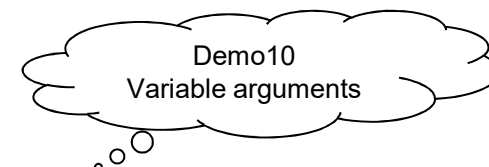
- **Decorator wrapper function must support variable arguments:**
  - To be able to work with different functions
  - Also need to add code to `new_function` so that it will accept arguments
- **Define general wrapper function using `*args` and `**kwargs`:**

```
def helloSolarSystem(original_function):
 def new_function(*args, **kwargs):
 original_function(*args, **kwargs)
 print("Hello, galaxy!")
 return new_function
```

- **Possible annotations:**

```
@helloSolarSystem
def hello(planet=None):
 @helloSolarSystem
 def goodbye(planets, moons, species):
```

- **Can also define wrapper for class method:**
  - But their first argument should be `self`





# Multiple Decorators

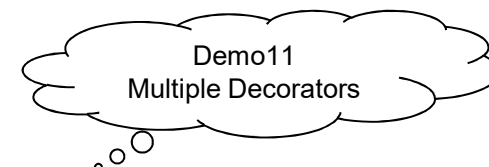
```
def p_decorate(func):
 def func_wrapper(name):
 return "<p>{0}</p>".format(func(name))
 return func_wrapper

def strong_decorate(func):
 def func_wrapper(name):
 return "{0}".format(func(name))
 return func_wrapper

def div_decorate(func):
 def func_wrapper(name):
 return "<div>{0}</div>".format(func(name))
 return func_wrapper

@div_decorate
@p_decorate
@strong_decorate
def get_text(name):
 return "Hello {0} how are you".format(name)

print(get_text("Mamaloe"))
```



# Class Decorators

- **Classes can also be decorated in Python:**
  - Just like functions can be decorated with other functions
- **Class decorators add required functionality:**
  - External to class implementation
- **Class decorators:**
  - Run at end of class statement to rebind a class name to a callable
  - Used to manage classes when they are created
  - Can be used to insert a layer of wrapper logic to manage instances

## Decorator definition:

```
def class_decorator(cls):

```

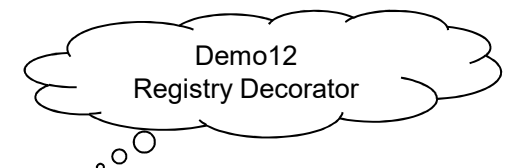
## Class definition:

```
@class_decorator
class My_Class:

```

## This is equivalent to :

```
My_Class = class_decorator(My_Class)
```



# Singleton Class

```
instances = {} #dictionary to hold class and their only instance

def getInstance(klass,*args): #this function is used by decorator
 if klass not in instances:
 instances[klass]=klass(*args)
 return instances[klass]

def singleton(klass): #decorator function
 def onCall(*args):
 return getInstance(klass,*args)
 return onCall

@singleton
class Star:
 def __init__(self,name):
 self.name=name

#A solar system should have only one star
sun = Star('Sun')
print sun.name
taurus = Star('Taurus')
print taurus.name
```

