Python

CSE 307 – Principles of Programming Languages

Stony Brook University

http://www.cs.stonybrook.edu/~cse307

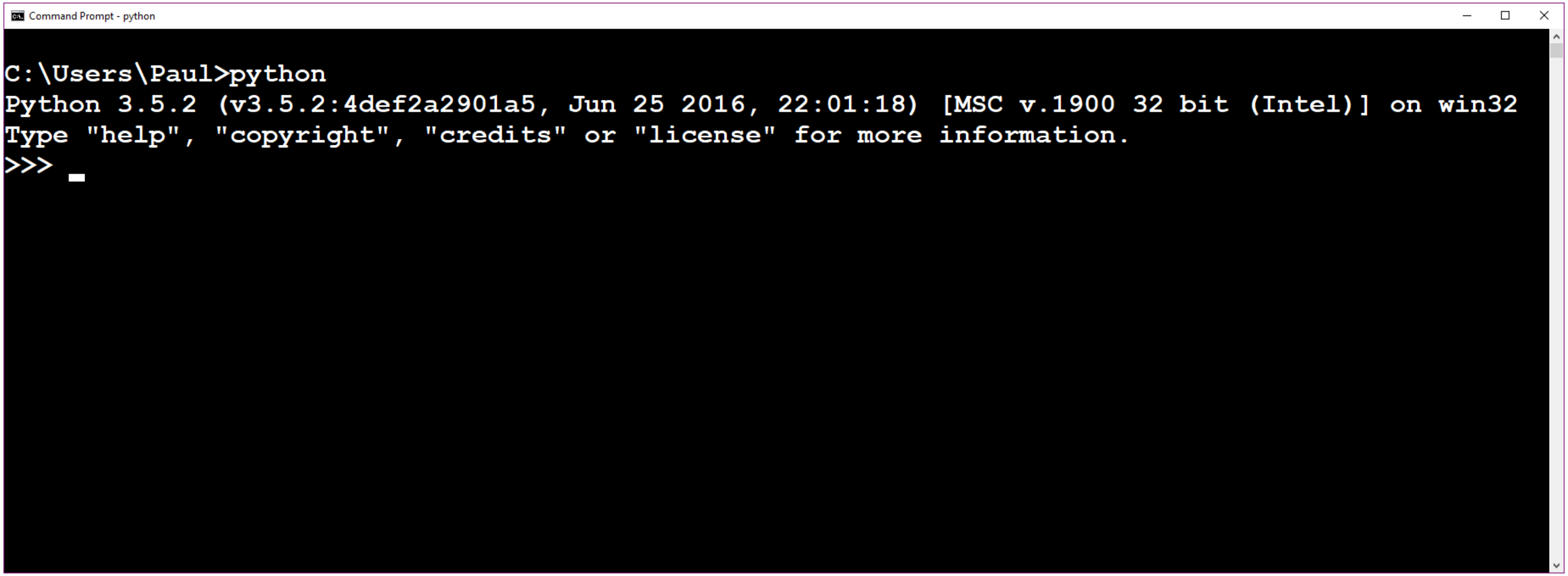
Python's History

- Created by Guido van Rossum in Netherlands in 1990
- Open source: http://www.python.org

Python 2.7x vs. Python 3.x

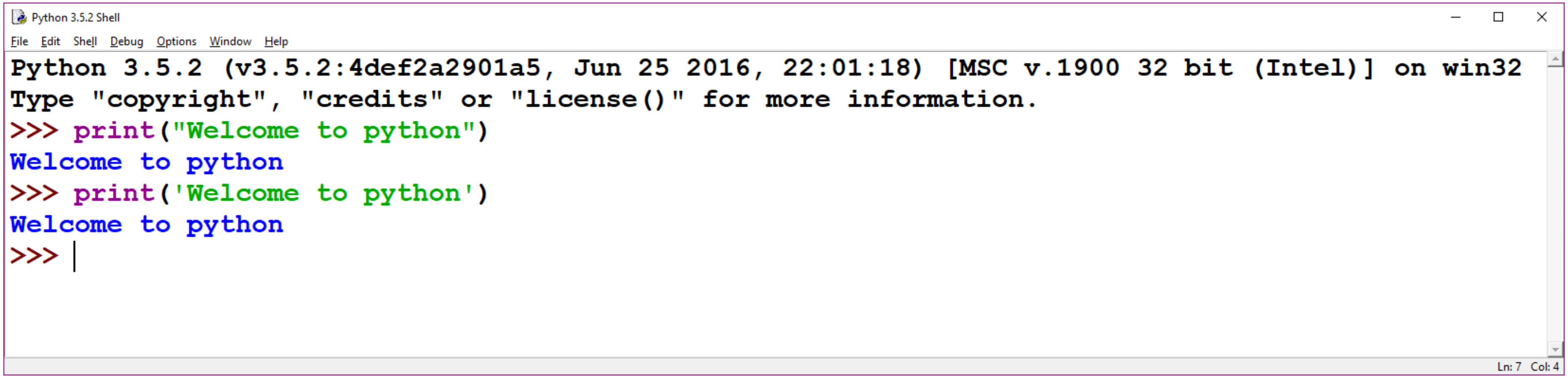
- Python 3.x is a newer version, but it is not backward compatible with Python 2.7x
  - That means if you write a program using Python 2, it may not work on Python 3.x
  - We use Python 3.x for homeworks

Launch Python



```
C:\Users\Paul>python
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Launch Python IDLE



```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Welcome to python")
Welcome to python
>>> print('Welcome to python')
Welcome to python
>>> 
```

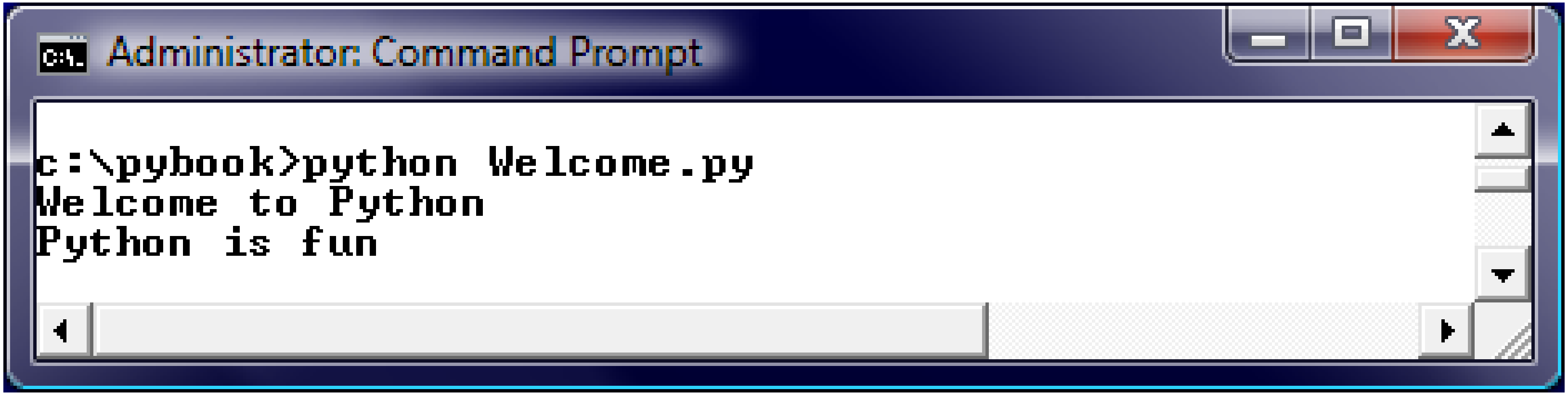
Editor, Command line interface, Debugger

Many other IDEs.

A Simple Python Program

```
# Welcome.py
# Display two messages
print("Welcome to Python")
print("Python is fun")

# Comment in Python
```

Run Python Script



```
c:\pybook>python Welcome.py
Welcome to Python
Python is fun
```

Python Example

- Dynamic Type inference:

```
# Assign a radius
radius = 20 # radius is now 20
# Compute area
area = radius * radius * 3.14159
# Display results
print("The area for the circle of radius " +
    str(radius) + " is " + str(area))
```

Reading Input from the Console

- Use the input function

```
variable = input("Enter a string: ");
```

- Use the eval function

```
variable = eval("51 + (54 * (3 + 2))");
print(variable);
321
```

Variables

```
# Compute the first area
radius = 1.0
area = radius * radius * 3.14159
print("The area is ", area, " for  radius ", radius)

# Compute the second area
radius = 2.0
area = radius * radius * 3.14159
print("The area is ", area, " for radius ", radius)
```

Types

- In Python 3, there is effectively no limit to how long an integer value can be - of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be as long as you need it to be
- The following strings can be prepended to an integer value to indicate a base other than 10:

`0b, oB`	Binary (base 2)
`0o, 0O` (zero + uppercase letter 'O')	Octal (base 8)
`0x, 0X`	Hexadecimal (base 16)

Types

```
>>> 10
10
>>> 0x10
16
>>> 0b10
2

>>> type(10)
<class 'int'>
>>> type(0o10)
<class 'int'>
>>> type(0x10)
<class 'int'>
```

Types

- The **`float`** type in Python designates a floating-point number: values are specified with a decimal point

https://docs.python.org/3.6/tutorial/floatingpoint.html

- Optionally, the character **`e`** or **`E`** followed by a positive or negative integer may be appended to specify scientific notation:

```
>>> 4.2
4.2
>>> type(4.2)
<class 'float'>
>>> 4.
4.0
>>> .2
0.2
```

```
>>> .4e7
4000000.0
>>> type(.4e7)
<class 'float'>
>>> 4.2e-4
0.00042
```

Types

- Float values as 64-bit “double-precision” values, according to the IEEE 754 standard (https://en.wikipedia.org/wiki/IEEE_754-2008_revision). The maximum value a floating-point number can have is approximately **1.8X10**^{308}. Python will indicate a number greater than that by the string **inf**:

  ```
  >>> 1.79e308
  1.79e+308
  >>> 1.8e308
  inf
  ```

- The closest a nonzero number can be to zero is approximately **5.0×10**$^{-324}$. Anything closer to zero than that is effectively zero:

  ```
  >>> 5e-324
  5e-324
  >>> 1e-325
  0.0
  ```

Types

- Floating point numbers are represented internally as binary (base-2) fractions
  - Most decimal fractions cannot be represented exactly as binary fractions, so in most cases the internal representation of a floating-point number is an approximation of the actual value
    - In practice, the difference between the actual value and the represented value is very small and should not usually cause significant problems

Expression

```
x = 1             # Assign 1 to variable x
radius = 1.0      # Assign 1.0 to variable radius

# Assign the value of the expression to x
x = 5 * (3 / 2) + 3 * 2
print(x)
           13.5
x = 5 * (3 // 2) + 3 * 2
print(x)
           11
```

Overflow

- When a variable is assigned a value that is too large (in size) to be stored, it causes overflow. For example, executing the following statement causes overflow:

```
>>>245.0 ** 1000000
OverflowError: 'Result too large'
```

Type Conversion and Rounding

- datatype(value) :

`int(4.5)` => 4

`float(4)` => 4.0

`str(4)` => '4'

`round(4.6)` => 5

`round(4.5)` => 4

`round(4.5)`=> 4 # in Python 3

`round(4.5)`=> 5 # in Python 2

https://docs.python.org/2/library/functions.html#round

https://docs.python.org/3/library/functions.html#round

Note: 2 vs 3

Built-in Functions and math Module

```
>>> max(2, 3, 4) # Returns a maximum number
4
>>> min(2, 3, 4) # Returns a minimum number
2
>>> round(3.51) # Rounds to its nearest integer
4
>>> round(3.4) # Rounds to its nearest integer
3
>>> abs(-3) # Returns the absolute value
3
>>> pow(2, 3) # Same as 2 ** 3
8
```

Function	Description	Example
fabs(x)	Returns the absolute value of the argument.	fabs(-2) is 2
ceil(x)	Rounds x up to its nearest integer and returns this integer.	ceil(2.1) is 3 ceil(-2.1) is -2
floor(x)	Rounds x down to its nearest integer and returns this integer.	floor(2.1) is 2 floor(-2.1) is -3
exp(x)	Returns the exponential function of x (e^x).	exp(1) is 2.71828
log(x)	Returns the natural logarithm of x.	log(2.71828) is 1.0
log(x, base)	Returns the logarithm of x for the specified base.	log10(10, 10) is 1
sqrt(x)	Returns the square root of x.	sqrt(4.0) is 2.0
sin(x)	Returns the sine of x. x represents an angle in radians.	sin(3.14159 / 2) is 1 sin(3.14159) is 0
asin(x)	Returns the angle in radians for the inverse of sine.	asin(1.0) is 1.57 asin(0.5) is 0.523599
cos(x)	Returns the cosine of x. x represents an angle in radians.	cos(3.14159 / 2) is 0 cos(3.14159) is -1
acos(x)	Returns the angle in radians for the inverse of cosine.	acos(1.0) is 0 acos(0.5) is 1.0472
tan(x)	Returns the tangent of x. x represents an angle in radians.	tan(3.14159 / 4) is 1 tan(0.0) is 0
fmod(x, y)	Returns the remainder of x/y as double.	fmod(2.4, 1.3) is 1.1
degrees(x)	Converts angle x from radians to degrees	degrees(1.57) is 90
radians(x)	Converts angle x from degrees to radians	radians(90) is 1.57

from math import fabs

or

import math

Strings and Characters

A string is a sequence of characters. *String* literals can be enclosed in matching *single quotes* (') or *double quotes* ("). Python does not have a data type for characters. A single-character string represents a character.

```
letter = 'A'     # Same as letter = "A"
numChar = '4'    # Same as numChar = "4"
message = "Good morning"
   # Same as message = 'Good morning'
```

Raw string literals

- A "raw string literal" is a slightly different syntax for a string literal, in which a backslash, \, is taken as meaning "just a backslash" (except when it comes right before a quote that would otherwise terminate the literal) -- no "escape sequences" to represent newlines, tabs, backspaces, form-feeds:

```
>>> print('\nsadasd')

sadasd
>>> print(r'\nsadasd')
\nsadasd
>>>
```

Functions ord and chr

```
>>> ch = 'a'
>>> ord(ch)
97
>>> chr(98)
'b'
```

The str Function

The **`str`** function can be used to convert a number into a string. For example,

```
>>> s = str(3.4) # Convert a float to string
>>> s
'3.4'
>>> s = str(3) # Convert an integer to string
>>> s
'3'
```

The String Concatenation Operator

You can use the + operator to add two numbers.

The + operator can also be used to concatenate (combine) two strings. Here are some examples:

```
>>> message = "Welcome " + "to " + "Python"
>>> message
'Welcome to Python'
>>> chapterNo = 1
>>> s = "Chapter " + str(chapterNo)
>>> s
'Chapter 1'
>>> s = "Chapter " + chapterNo
TypeError: Can't convert 'int' object to str implicitly
```

Introduction to Objects and Methods

- In Python, all data—including numbers and strings—are actually objects.
- An object is an entity. Each object has an id and a type. Objects of the same kind have the same type. You can use the **id** function and **type** function to get these information for an object.

Object Types and Ids

The **id** and **type** functions are rarely used in programming, but they are good pedagogical tools for understanding objects.

```
>>> n = 3  # n is an int
>>> id(n)
505408904
>>> type(n)
<class ’int’>
>>> f = 3.0  # f is a float
>>> id(f)
26647120
```

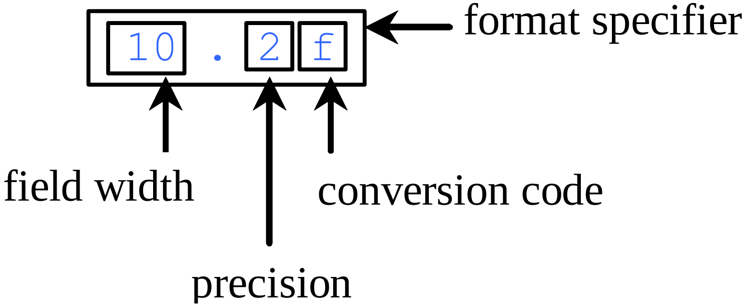
```
>>> type(f)
<class ’float’>
>>> s = "Welcome"
>>> id(s)
36201472
>>> type(s)
<class ’str’>
```

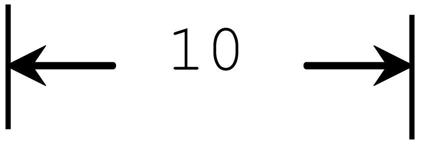
str Object Methods

```
>>> s = "Welcome"
>>> s1 = s.lower() # Invoke the lower method
>>> s1
'welcome'

>>> s2 = s.upper() # Invoke the upper method
>>> s2
'WELCOME'
```

Formatting Floating-Point Numbers

```
print(format(57.467657, '10.2f'))
print(format(12345678.923, '10.2f'))
print(format(57.4, '10.2f'))
print(format(57, '10.2f'))
```





```
????57.47
12345678.9
????57.40
????57.00
```

Blocks

- Python 3 uses indentation of 4 spaces for blocks
    - Tabs should be used solely to remain consistent with code that is already indented with tabs

https://www.python.org/dev/peps/pep-0008/#tabs-or-spaces

"*Python 3 disallows mixing the use of tabs and spaces for indentation.*"

if...else Example

```
from math import pi
if radius >= 0:
    area = radius * radius * pi
    print("The area for the ",
        "circle of radius ",
        radius, " is ", area)
else:
    print("Negative input")
```

Multiple Alternative if Statements

```
if score >= 90.0:
    grade = 'A'
else:
  if score >= 80.0:
      grade = 'B'
  else:
      if score >= 70.0:
          grade = 'C'
      else:
          if score >= 60.0:
              grade = 'D'
          else:
              grade = 'F'
```

(a)

Equivalent

This is better

```
if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'
```

(b)

Loops

```
# Initialize loop-control variable
i = initialValue
while i < endValue:
      # Loop body
      ...
      i+=1 # Adjust loop-control variable

for i in range(initialValue, endValue):
      # Loop body
```

range(a, b)

```
for i in range(4, 8):
    print(i)
```

```
4
5
6
7
```

range(b)

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

range(a, b, step)

```
for v in range(3, 9, 2):
    print(v)
3
5
7
```

Functions

```
def sum(i1, i2):
    ''' This is the doc '''
    result = 0
    for i in range(i1, i2):
        result += i
    return result
def main():
    print("Sum from 1 to 10 is", sum(1, 10))
    print("Sum from 20 to 37 is", sum(20, 37))
    print("Sum from 35 to 49 is", sum(35, 49))
main() # Call the main function
```

Classes

```
import math
class Circle:
    # Construct a circle object
    def __init__(self, radius = 1):
        self.radius = radius
    def getPerimeter(self):
        return 2 * self.radius * math.pi
    def getArea(self):
        return self.radius * self.radius * math.pi
    def setRadius(self, radius):
        self.radius = radius
    def __str__(self):
        return "Circle: radius=" + str(radius)
```

```
from Circle import Circle

def main():
    # Create a circle with radius 1
    circle1 = Circle()
    print("The area of the circle of radius", circle1.radius,
        "is", circle1.getArea())
    # Create a circle with radius 25
    circle2 = Circle(25)
    print("The area of the circle of radius", circle2.radius,
        "is", circle2.getArea())
    # Create a circle with radius 125
    circle3 = Circle(125)
    print("The area of the circle of radius", circle3.radius,
        "is", circle3.getArea())
    # Modify circle radius
    circle2.radius = 100
    print("The area of the circle of radius", circle2.radius,
        "is", circle2.getArea())

main() # Call the main function
```

Inheritance

```
from GeometricObject import GeometricObject
import math
class Circle(GeometricObject):
    def __init__(self, radius):
        super().__init__()
        self.__radius = radius
    def getRadius(self):
        return self.__radius
    def setRadius(self, radius):
        self.__radius = radius
    def getArea(self):
        return self.__radius * self.__radius * math.pi
    def getDiameter(self):
        return 2 * self.__radius
    def getPerimeter(self):
        return 2 * self.__radius * math.pi
    def printCircle(self):
        print(self.__str__() + " radius: " +
            str(self.__radius))
```

Adding fields to Objects dynamically

```
class Employee:
    pass

# Create an empty employee record
john = Employee()

# Add the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Underscores

- Single and double underscores have a meaning in Python variable and method names
  - Some of that meaning is merely by convention and intended as a hint to the programmer—and some of it is enforced by the Python interpreter.

Underscores

- Single Leading Underscore: **`_var`**

https://pep8.org/#descriptive-naming-styles

  - Naming convention indicating a name is meant for internal/private use
    - Python does not have strong distinctions between "private" and "public" variables like Java does
  - Generally not enforced by the Python interpreter (except in wildcard imports – see next)

Underscores

- Single Leading Underscore: **`_var`**

```
# This is my_module.py:
def _internal_func():
    return 42
>>> from my_module import *
>>> _internal_func()
NameError: "name '_internal_func' is not defined"
```

- Wildcard imports should be avoided as they make it unclear which names are present in the namespace.
  - It’s better to stick to regular imports for the sake of clarity.

Underscores

- Single Trailing Underscore: `var_`
  - Used by convention to avoid naming conflicts with Python keywords. Sometimes the most fitting name for a variable is already taken by a keyword. Therefore names like `class` cannot be used as variable names in Python.
  - You can append a single underscore to break the naming conflict:

```
>>> def make_object(name, class):
SyntaxError: "invalid syntax"
>>> def make_object(name, class_):
...     pass
>>>
```

Underscores

- Double Leading Underscore: **`__var`**
  - causes the Python interpreter to rewrite the attribute name in order to avoid naming conflicts in subclasses (called ***name mangling***—the interpreter changes the name of the variable in a way that makes it harder to create collisions when the class is extended later)

Underscores

- Double Leading Underscore: `__var`

```
class Test:
    def __init__(self):
        self.foo = 11
        self._bar = 23
        self.__baz = 23
>>> t = Test()
>>> dir(t)
['_Test__baz', '__class__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_bar', 'foo']
```

Underscores

```
class ExtendedTest(Test):
    def __init__(self):
        super().__init__()
        self.foo = 'overridden'
        self._bar = 'overridden'
        self.__baz = 'overridden'
>>> t2 = ExtendedTest()
>>> t2.foo
'overridden'
>>> t2._bar
'overridden'
>>> t2.__baz
AttributeError: "'ExtendedTest' object has no attribute '__baz'"
>>> t2._ExtendedTest__baz
'overridden'
```

- __baz got turned into _ExtendedTest__baz to prevent accidental modification
- The original _Test__baz is also still around:

```
>>> t2._Test__baz
42
```

Underscores

- Not in the class itself:

```
class ManglingTest:
    def __init__(self):
        self.__mangled = 'hello'

    def get_mangled(self):
        return self.__mangled

>>> ManglingTest().get_mangled()
'hello'

>>> ManglingTest().__mangled
AttributeError: "'ManglingTest' object has no
attribute '__mangled'"

>>> ManglingTest()._ManglingTest__mangled
'hello'
```

Underscores

- Not in the class itself:

```
class MangledMethod:
    def __method(self):
        return 42

    def call_it(self):
        return self.__method()

>>> MangledMethod().__method()
AttributeError: "'MangledMethod' object has no
attribute '__method'"

>>> MangledMethod().call_it()
42

>>> MangledMethod()._MangledMethod__method()
42
```

Underscores

- Weird:

```
_MangledGlobal__mangled = 23

class MangledGlobal:
    def test(self):
        return __mangled

>>> MangledGlobal().test()
23
```

Underscores

- Double underscores are often referred to as “dunders” in the Python community
  - The reason is that double underscores appear quite often in Python code and to avoid fatiguing their jaw muscles Pythonistas often shorten “double underscore” to “dunder.”
  - It’s like a secret handshake for Python developers
- Name mangling is not applied if a name starts and ends with double underscores

```
class PrefixPostfixTest:
    def __init__(self):
        self.__bam__ = 42

>>> PrefixPostfixTest().__bam__
42
```

Underscores

- Names that have **both leading and trailing double underscores** are reserved for special use in the language
  - This rule covers things like **`__init__`** for object constructors, or **`__call__`** to make an object callable
  - Indicates special methods defined by the Python language.
    - Avoid this naming scheme for your own attributes.
  - These dunder methods are often referred to as ***magic methods***

Underscores

- Single Underscore has 2 uses:
  - _ used as a name for temporary or insignificant variables ("don't care").

```
>>> for _ in range(32):
...     print('Hello, World.')
```

Underscores

- REPLs that represents the result of the last expression evaluated by the interpreter

```
>>> 20 + 3
23
>>> _
23
>>> print(_)
23
>>> list()
[]
>>> _.append(1)
>>> _.append(2)
>>> _.append(3)
>>> _
[1, 2, 3]
```

Exceptions

```
from GeometricObject import GeometricObject
import math
class Circle(GeometricObject):
    def __init__(self, radius):
        super().__init__()
        self.setRadius(radius)
    def setRadius(self, radius):
        if radius < 0:
            raise RuntimeError("Negative radius")
        else:
            self.__radius = radius
```

The str Class

Creating Strings

```
s1 = str()          # Create an empty string
s2 = str("Welcome") # Create a string Welcome
```

Python provides a simple syntax for creating string using a string literal. For example,

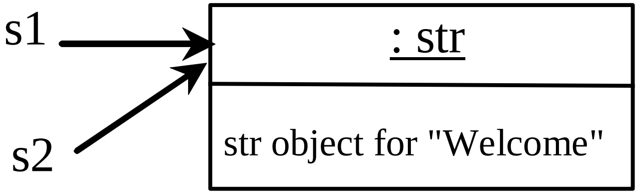
```
s1 = ""               # Same as s1 = str()
s2 = "Welcome"
                # Same as s2 = str("Welcome")
```

Strings are Immutable

A string object is immutable. Once it is created, its contents cannot be changed. To optimize performance, Python uses one object for strings with the same contents.

- both s1 and s2 refer to the same string object.

```
>>> s1 = "Welcome"
>>> s2 = "Welcome"
>>> id(s1)
505408902
>>> id(s2)
505408902
```



Functions for str

```
>>> s = "Welcome"
>>> len(s)
7
>>> max(s)
o
>>> min(s)
W
```

The +, *, [:], and in Operators

```
>>> s1 = "Welcome"
>>> s2 = "Python"
>>> s3 = s1 + " to " + s2
>>> s3
'Welcome to Python'
>>> s4 = 2 * s1
>>> s4
'WelcomeWelcome'
>>> s1[3 : 6]
'com'
>>> 'W' in s1
True
>>> 'X' in s1
False
```

Negative Index

```
>>> s1 = "Welcome"
>>> s1[-1]
'e'
>>> s1[-3 : -1]
'om'
```

The in and not in Operators

```
>>> s1 = "Welcome"
>>> "come" in s1
True
>>> "come" not in s1
False
>>>
```

Foreach Loops

```
for ch in string:
    print(ch)


for i in range(0, len(s), 2):
    print(s[i])
```

Comparing Strings

```
>>> s1 = "green"
>>> s2 = "glow"
>>> s1 == s2
False
>>> s1 != s2
True
>>> s1 > s2
True
>>> s1 >= s2
True
>>> s1 < s2
False
>>> s1 <= s2
False
```

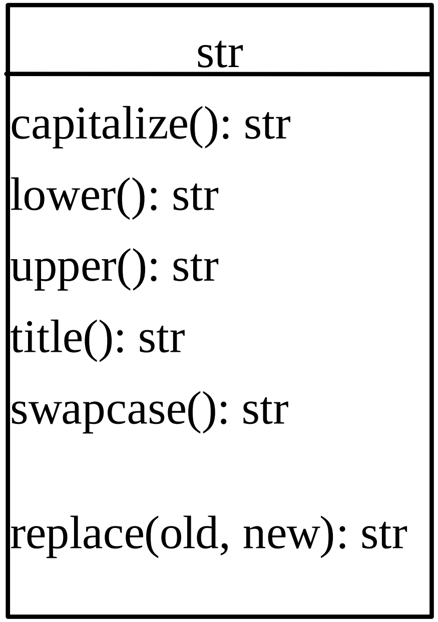
Testing Characters in a String

str	
isalnum(): bool	Return True if all characters in this string are alphanumeric and there is at least one character.
isalpha(): bool	Return True if all characters in this string are alphabetic and there is at least one character.
isdigit(): bool	Return True if this string contains only number characters.
isidentifier(): bool	Return True if this string is a Python identifier.
islower(): bool	Return True if all characters in this string are lowercase letters and there is at least one character.
isupper(): bool	Return True if all characters in this string are uppercase letters and there is at least one character.
isspace(): bool	Return True if this string contains only whitespace characters.

Searching for Substrings

str	
endswith(s1: str): bool	Returns True if the string ends with the substring s1.
startswith(s1: str): bool	Returns True if the string starts with the substring s1.
find(s1): int	Returns the lowest index where s1 starts in this string, or -1 if s1 is not found in this string.
rfind(s1): int	Returns the highest index where s1 starts in this string, or -1 if s1 is not found in this string.
count(subtring): int	Returns the number of non-overlapping occurrences of this substring.

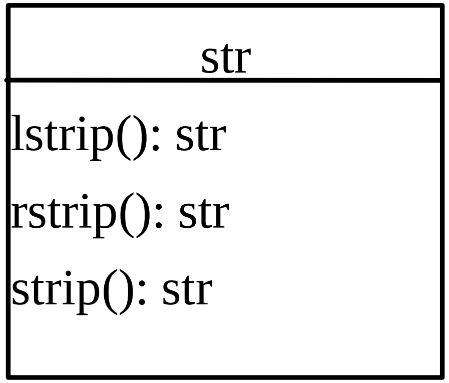
Converting Strings



str

capitalize(): str	Returns a copy of this string with only the first character capitalized.
lower(): str	Returns a copy of this string with all characters converted to lowercase.
upper(): str	Returns a copy of this string with all characters converted to uppercase.
title(): str	Returns a copy of this string with the first letter capitalized in each word.
swapcase(): str	Returns a copy of this string in which lowercase letters are converted to uppercase and uppercase to lowercase.
replace(old, new): str	Returns a new string that replaces all the occurrence of the old string with a new string.

Stripping Whitespace Characters



str

lstrip(): str

rstrip(): str

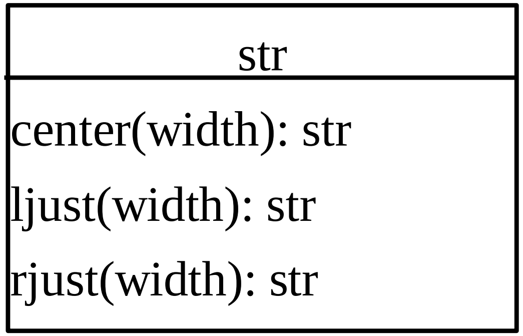
strip(): str

Returns a string with the leading whitespace characters removed.

Returns a string with the trailing whitespace characters removed.

Returns a string with the starting and trailing whitespace characters removed.

Formatting Strings



str	
center(width): str	Returns a copy of this string centered in a field of the given width.
ljust(width): str	Returns a string left justified in a field of the given width.
rjust(width): str	Returns a string right justified in a field of the given width.

Creating Lists

Creating list using the list class

```
list1 = list() # Create an empty list
list2 = list([2, 3, 4]) # Create a list with elements 2, 3, 4
list3 = list(["red", "green", "blue"]) # Create a list with strings
list4 = list(range(3, 6)) # Create a list with elements 3, 4, 5
list5 = list("abcd") # Create a list with characters a, b, c, d
```

For convenience, you may create a list using the following syntax:

```
list1 = [] # Same as list()
list2 = [2, 3, 4] # Same as list([2, 3, 4])
list3 = ["red", "green"] # Same as list(["red", "green"])
```

list Methods

list	
append(x: object): None	Add an item x to the end of the list.
insert(index: int, x: object): None	Insert an item x at a given index. Note that the first element in the list has index 0.
remove(x: object): None	Remove the first occurrence of the item x from the list.
index(x: object): int	Return the index of the item x in the list.
count(x: object): int	Return the number of times item x appears in the list.
sort(): None	Sort the items in the list.
reverse(): None	Reverse the items in the list.
extend(l: list): None	Append all the items in L to the list.
pop([i]): object	Remove the item at the given position and return it. The square bracket denotes that parameter is optional. If no index is specified, list.pop() removes and returns the last item in the list.

Functions for lists

```
>>> list1 = [2, 3, 4, 1, 32]
>>> len(list1)
5
>>> max(list1)
32
>>> min(list1)
1
>>> sum(list1)
42
>>> import random
>>> random.shuffle(list1) # Shuffle the items in the list
>>> list1
[4, 1, 2, 32, 3]
```

The +, *, [:], and in Operators

```
>>> list1 = [2, 3]
>>> list2 = [1, 9]
>>> list3 = list1 + list2
>>> list3
[2, 3, 1, 9]
>>> list3 = 2 * list1
>>> list3
[2, 3, 2, 3]
>>> list4 = list3[2 : 4]
>>> list4
[2, 3]
```

The +, *, [:], and in Operators

```
>>> list1 = [2, 3, 5, 2, 33, 21]
>>> list1[-1]
21
>>> list1[-3]
2
>>> list1 = [2, 3, 5, 2, 33, 21]
>>> 2 in list1
True
>>> list1 = [2, 3, 5, 2, 33, 21]
>>> 2.5 in list1
False
```

Comparing Lists

```
>>>list1 = ["green", "red", "blue"]
>>>list2 = ["red", "blue", "green"]
>>>list2 == list1
False
>>>list2 != list1
True
>>>list2 >= list1
True
>>>list2 > list1
True
>>>list2 < list1
False
>>>list2 <= list1
False
```

Splitting a String to a List

```
items = "Welcome to CSE307".split()
print(items)
['Welcome', 'to', 'CSE307']

items = "34#13#78#45".split("#")
print(items)
['34', '13', '78', '45']
```

Pass-by-Value Example

```
def main():
    x = 1 # x represents an int value
    y = [1, 2, 3] # y represents a list
    m(x, y) # Invoke f with arguments x and y
    print("x is " + str(x))
    print("y[0] is " + str(y[0]))
def m(number, numbers):
    number = 1001 # Assign a new value to number
    numbers[0] = 5555 # Assign a new value to numbers[0]
main()
```

Binary Search

```
# Use binary search to find the key in the list
def binarySearch(lst, key):
    low = 0
    high = len(lst) - 1
    while high >= low:
        mid = (low + high) // 2
        if key < lst[mid]:
            high = mid - 1
        elif key == lst[mid]:
            return mid
        else:
            low = mid + 1
    # Now high < low, key not found
    return -low - 1
```

Selection Sort

```
def selectionSort(lst):
    for i in range(0, len(lst) - 1):
        # Find the minimum in the lst[i..len(lst)-1]
        currentMin = lst[i]
        currentMinIndex = i
        for j in range(i + 1, len(lst)):
            if currentMin > lst[j]:
                currentMin = lst[j]
                currentMinIndex = j
        # Swap lst[i] with lst[currentMinIndex] if necessary
        if currentMinIndex != i:
            lst[currentMinIndex] = lst[i]
            lst[i] = currentMin
    return lst
```

Write to a File

```
outfile = open("test.txt", "w")
outfile.write("Welcome to Python")
```

file	
read([number: int]): str	Returns the specified number of characters from the file. If the argument is omitted, the entire remaining contents are read.
readline(): str	Returns the next line of file as a string.
readlines(): list	Returns a list of the remaining lines in the file.
write(s: str): None	Writes the string to the file.
close(): None	Closes the file.

Testing File Existence

```
import os.path
if os.path.isfile("Presidents.txt"):
    print("Presidents.txt exists")
```

Write/Read in/from File

```
def main():
    # write
    w = open("a.txt", "w")
    w.write("de")
    w.close()
    # read
    r = open("a.txt", "r")
    for line in r:
        print(line)
    r.close()
main()
```

Tuples

```
t1 = () # Create an empty tuple
t2=(1,3,5) # Create a set with three elements
# Create a tuple from a list
t3 = tuple([2*x for x in range(1,5)])
# Create a tuple from a string
t4 = tuple("abac") # t4 is ['a', 'b', 'a', 'c']
```

- Tuples vs. lists: you cannot modify a tuple!

List Comprehensions

- List comprehensions are a concise way to create lists

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

same with:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
```

but shorter

List Comprehensions

```
>>> vec = [-4, -2, 0, 2, 4]
# create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
# filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
# apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
```

List Comprehensions

- A list comprehension consists of brackets containing an expression followed by a **`for`** clause, then zero or more **`for`** or **`if`** clauses
  - the result will be a new list resulting from evaluating the expression in the context of the **`for`** and **`if`** clauses which follow it
  - example: combines the elements of two lists if they are not equal

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

List Comprehensions

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

is the same with:

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

List Comprehensions

```
# create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

# flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List Comprehensions

```
# Nested List Comprehensions
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
>>> [ [row[i] for row in matrix]
              for i in range(len(matrix[0]))]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

all and any

- **`all(iterable)`** returns **`True`** if all elements of the **`iterable`** are true (or if the **`iterable`** is empty)
    - The internal implementation:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

all and any

- **`any(iterable)`** returns **`True`** if any element of the **`iterable`** is true. If the **`iterable`** is empty, return **`False`**.
  - The internal implementation:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

all and any Example

```
def is_prime(element):
    if element == 2:
        return True
    elif element <= 1 or element % 2 == 0:
        return False
    else:
        return all(element%i for i
                      in range(3,element,2))

myList = [4, 5, 9, 12]
if not any(is_prime(x) for x in myList):
    print("The list did not contain a prime")
else:
    print("The list contains a prime")
```

Python’s iterator protocol

- Objects that support the __iter__ and __next__ dunder methods automatically work with for-in loops

```
class Repeater:
    def __init__(self, value):
        self.value = value
    def __iter__(self):
        return RepeaterIterator(self)
class RepeaterIterator:
    def __init__(self, source):
        self.source = source
    def __next__(self):
        return self.source.value
repeater = Repeater('Hello')
for item in repeater:
    print(item)
Hello
Hello
...
```

Python’s iterator protocol

```
class BoundedRepeater:
    def __init__(self, value, max_repeats):
        self.value = value
        self.max_repeats = max_repeats
        self.count = 0
    def __iter__(self):
        self.value
    def __next__(self):
        if self.count >= self.max_repeats:
            raise StopIteration
        self.count += 1
        return self.value
>>> repeater = BoundedRepeater('Hello', 3)
>>> for item in repeater:
        print(item)
Hello
Hello
Hello
```

Sets

```
# Create an empty set
s1 = set()

# Create a set with three elements
s2 = {1, 3, 5}

# Create a set from a list
s3 = set([1, 3, 5])

# Create a set from a list
s4 = set([x * 2 for x in range(1, 10)])

# Create a set from a string
s5 = set("abac") # s5 is {'a', 'b', 'c'}
```

Manipulating and Accessing Sets

```
>>> s1 = {1, 2, 4}
>>> s1.add(6)
>>> s1
{1, 2, 4, 6}
>>> len(s1)
4
>>> max(s1)
6
>>> min(s1)
1
>>> sum(s1)
13
>>> 3 in s1
False
>>> s1.remove(4)
>>> s1
{1, 2, 6}
>>>
```

Equality Test

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 4, 2}
>>> s1 == s2
True
>>> s1 != s2
False
>>>
```

Subset and Superset

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 4, 5, 2, 6}
>>> s1.issubset(s2) # s1 is a subset of s2
True
>>>
```

```
>>> s2.issuperset(s1) #s2 is a superset of s1
True
>>>
```

Comparison Operators

- Note that it makes no sense to compare the sets using the conventional comparison operators (>, >=, <=, <), because the elements in a set are not ordered.
- However, these operators have special meaning when used for sets.

s1 > s2 returns true is s1 is a proper superset of s2.

s1 >= s2 returns true is s1 is a superset of s2.

s1 < s2 returns true is s1 is a proper subset of s2.

s1 <= s2 returns true is s1 is a subset of s2.

Set Operations (union, |)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.union(s2)
{1, 2, 3, 4, 5}

# same with:
>>> s1 | s2
{1, 2, 3, 4, 5}
```

Set Operations (intersection, &)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.intersection(s2)
{1}

# same with:
>>> s1 & s2
{1}
```

Set Operations (difference, -)

```
>>> s1 = {1, 2, 4}
>>> s2 = {1, 3, 5}
>>> s1.difference(s2)
{2, 4}

>>> s1 - s2
{2, 4}
```

Creating a Dictionary

```
# Create an empty dictionary
dictionary = {}
# Create a dictionary
dictionary = {"john":40,
    "peter":45}
```

Looping Entries

```
for key in dictionary:
    print(key + ":" +
        str(dictionary[key]))
```

Lambda Expressions

- Small anonymous functions
  - a function can return a function

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

Standard Library

- Operating System Interface:

```
>>> import os

# Return the current working directory
>>> os.getcwd()
'C:\\Python35'

# Run the command mkdir
>>> os.system('mkdir today')
0
```

Standard Library

- Operating System Interface:

```
>>> import shutil

>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'

>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

Standard Library

- String Pattern Matching Interface:

```
>>> import re

>>> re.findall(r'\bf[a-z]*',
     'which foot or hand fell fastest')

['foot', 'fell', 'fastest']
```

Standard Library

- Mathematics:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'

# sampling without replacement
>>> random.sample(range(100), 10)
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]

>>> random.random()     # random float
0.17970987693706186
```

Standard Library

- Mathematics:

```
>>> import statistics

>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572

>>> statistics.median(data)
1.25

>>> statistics.variance(data)
1.3720238095238095
```

Standard Library

- Internet Access:

```
>>> from urllib.request import urlopen

>>> with urlopen('http://www.cs.stonybrook.edu') as response:
    for line in response:
        print(line)
```

Standard Library

- Dates and Times:

```
>>> from datetime import date

>>> now = date.today()
>>> now

>>> birthday = date(2000, 5, 23)

>>> age = now - birthday

>>> age.days
```

Standard Library

- Data Compression:

```
>>> import zlib
>>> s = b'data archiving and compression'
```

# A prefix of 'b' means that the chars are encoded in byte type
# may only contain ASCII characters

```
>>> t = zlib.compress(s)
>>> zlib.decompress(t)
b'data archiving and compression'
>>> zlib.crc32(s)
3701065259
```

Standard Library

- Testing:
  - doctest: scans a module and validate tests embedded in a program’s docstrings

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.
    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()  # automatically validate the embedded tests
```

Standard Library

- Testing:
  - unittest: comprehensive set of tests to be maintained in a separate file

```
import unittest
class TestStatisticalFunctions(unittest.TestCase):
    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

Standard Library

- Logging:

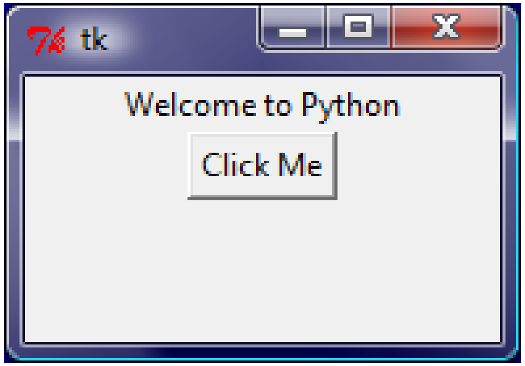
```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
logging.getLogger().setLevel('INFO')
```

  - by default, informational and debugging messages are suppressed:

Level	Numeric value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

Python GUIs with tkinter

```
from tkinter import * # Import tkinter

root = Tk() # Create a root window

# Create a label
label = Label(root, text = "Welcome to Python")

# Create a button
button = Button(root, text = "Click Me")

label.pack() # Display the label in the window
button.pack() # Display the button in the window

root.mainloop() # Create an event loop
```



What else?

- Lots:
  - The Python Standard Library: built-in functions, collections, and many modules: https://docs.python.org/3/library/index.html#library-index
  - Installing Python Modules: pip, virtual environments https://docs.python.org/3/installing/index.html#installing-index
  - The Python Language Reference: the syntax and "core semantics" https://docs.python.org/3/reference/index.html#reference-index