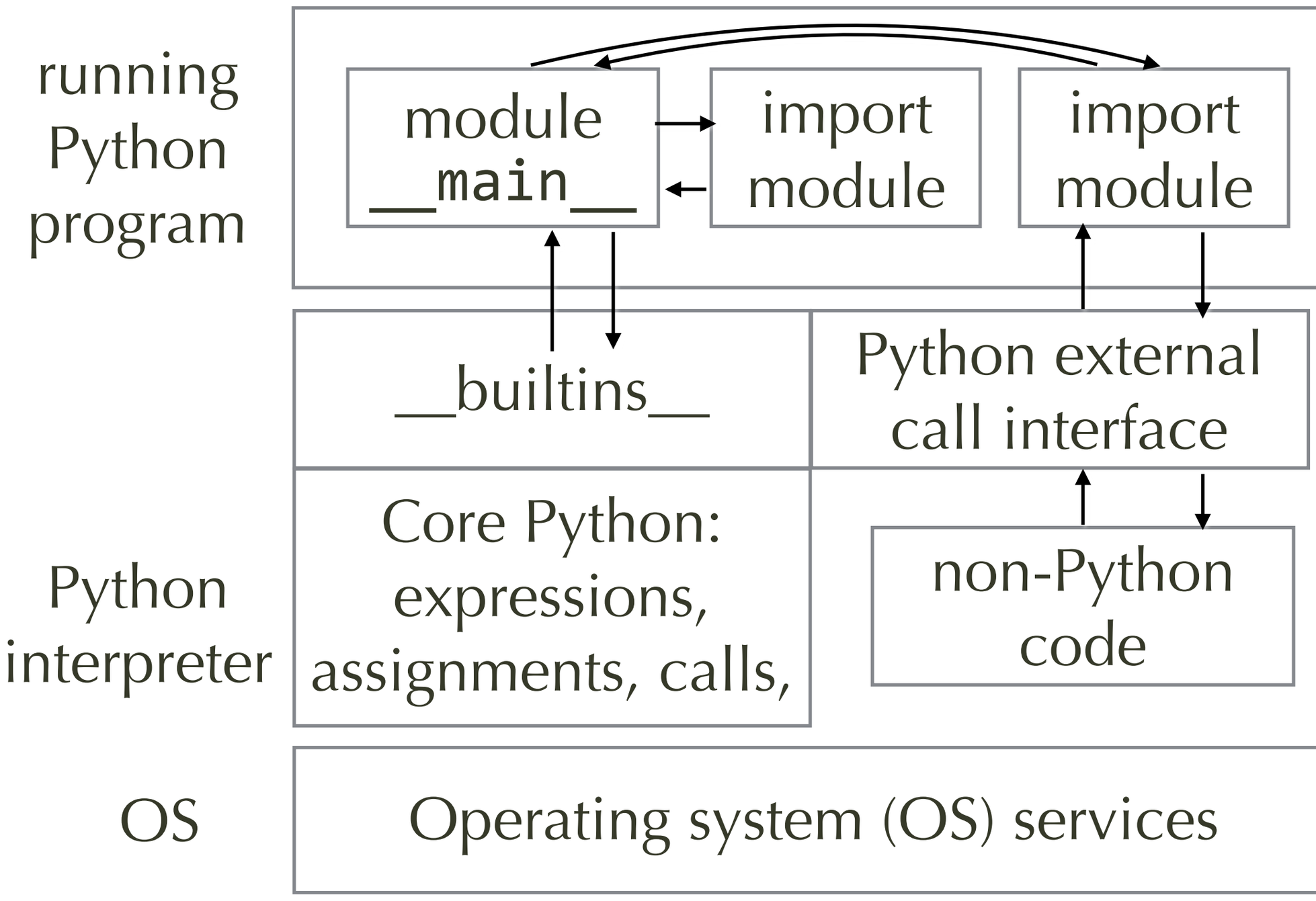
The Python Standard Library: part 1

Prof. Pai H. Chou
National Tsing Hua University

Outline

- Standard Library
  - https://docs.python.org/3/library/
  - Built-ins vs. Modules and Packages
- Built-ins
  - functions, constants, data types, exceptions
- Standard Modules
  - text, binary data, data types, numeric, functional programming, files, data persistence, compression, format, cryptography, operating systems, concurrent execution, networking, multimedia, internationalization, programming, user interface, ...

Python features



Built-ins functions

(as of Python 3.7)

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

eval(s)

- "evaluates" a string as Python source code for an expression

```
>>> s = '2 + 3'
>>> eval(s)
5
>>> t = "len(['a', 'b', 'c', 'd'])"
>>> eval(t)
4
>>> eval('hello world')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    hello world
               ^
SyntaxError: unexpected EOF while parsing
```

What name spaces does eval() have access to?

- By default, same as where your program is executing!

```
>>> a = 3
>>> eval('a + 4')
7
```

- May want to limit the name space
  - important if evaluating string from user!!

```
>>> a = 3
>>> eval('a + 4', {'a': 25} )
29
```

  - You can specify your own dict for symbol table

exec(s)

- executes string as Python source code!
  - as statements in the context of exec() statement
- `exec()` vs. `eval()`
  - `exec`: executes statements => no return value
  - `eval`: evaluates expression => has return value
  - Both can take optional global dict and local dict for controlling name space

Example use of exec()

- want a function to load a module whose name is not known until runtime
  - the name of the module is a fixed module name! it cannot be a variable or parameter; it is not a string
  - cannot say
    ```
    def my_import_func(moduleName)
        import moduleName
    ```
    - ModuleNotFoundError: No module named 'moduleName'
- Solution: create the string for import then exec it
  - ```
    def my_import_func(moduleName)
        exec(f'import {moduleName}')
    ```
  - To call, pass the module as a string e.g., `my_import_func('random')`
    ```

eval(s, globaldict, localdict)

- Global dict has `__builtins__` implicitly

```
>>> a = 3
>>> eval('abs(a + 4)', {'a': -25} )
21
```

  - to block out builtins, `{'__builtins__': None}`

- Local dict are searched first

  - can provide mapping `{'sin': math.sin }` (after importing math)

```
>>> import math
>>> eval('sin(a + 4)', {'__builtins__': None, 'sin': math.sin})
-0.836655638536056
```

compile(s)

- translates text into a callable object
- once compiled, can exec or evaluate it just like text
- more efficient than text, which requires additional processing (parsing, code generation) before running

Built-in constants

- strictly speaking, "runtime constants"
- Keywords: `False`, `True`, `None`
- `NotImplemented`
  - return value for special methods that are not implemented
  - Not the same as NotImplementedError exception!
- `__debug__`
  - True if Python interpreter not started with -o option

Built-in Types

https://docs.python.org/3/library/stdtypes.html

- bool
- int, float, complex
- dict
  - dictview
- list, tuple, range
- set, frozenset
- str
- bytes, bytearray, memoryview
- iterator
- generator
- function
- module
- various exceptions

Search order of built-in names

- Built-in names are searched **after** globals
- Could redefine built-in names in global space
  - => because they are not keywords!

```
>>> len([1,2,3])   # call the built-in function
3
>>> len = 'xyz'    # override built-in len with a new global name
>>> print(len+'ABC')
xyzABC
>>> len([1,2,3])   # try to call len as the built-in function
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
>>> len = __builtins__.len  # restore the built-in len
>>> len([1,2,3])
3
```

built-in functions vs. globals

- built-in functions can be *shadowed*
  - possible to define open, len, range, float, int, ...
  - however, this can be confusing and error prone
- Shadowed built-in names are still accessible
  - (e.g., delete the global to unshadow, or define the global to the **`__builtins__.`**name again

Example of confusing code

```
>>> max, min = min, max
>>> L = [1, 7, 3, 4, 6]
>>> print(max(L), min(L))
1 7
```

- source code becomes misleading, because max and min mean opposite!

- Don't try to change built-ins!

```
>>> __builtins__.len = 3
>>> len([1, 7, 3, 4, 6])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Modules in Python

- external python programs that can be imported
  - standard library (e.g., sys, os, math, random, string)
  - your own `.py` or `.pyc` file
  - custom-installed (e.g., tensorflow, flask, ...) using `pip`
    - Search order for importing is in `sys.path`
- Why modules?
  - reuse instead of re-inventing
  - Keep code organized and more manageable

What is in a module?

- Content
  - documentation string
  - variables and constants
  - functions definitions
  - classes definitions
  - statements (e.g., test cases, assignment, if-else, calls, ...)
- Object naming
  - explicitly by programmer or implicitly as special names

Where are the modules?

```
>>> import sys
>>> sys.path
['', '/usr/local/lib/python37.zip', '/usr/local/lib/python3.7/
site-packages']  # your result may vary
```

- your own paths may vary!
- `''` means the current directory where you start the python command
- Search order for the modules
  - look in the current directory first;
  - if not found, look in the next directory on the list

importing vs. loading

- Load
  - executes the given module
- Import: 2 steps
  - load the module, if it has not been imported before
  - add the imported symbols to the current name space
  - => if a module has been imported before, it will not be executed multiple times!

Standard modules

https://docs.python.org/3/library/

- Data types
- Numeric
- Functional Programming
- Text Processing
- File access, data persistence,, format
- Compression, cryptography
- OS service, concurrent execution
- Networking, HTML, XML
- Multimedia, internationalization
- Framework for graphic, GUI, command-line
- Development, debugging, packaging, runtime, language

Data types modules

https://docs.python.org/3/library/datatypes.html

- datetime — Basic
- calendar — General
- collections — Containers
- collections.abc — abstract base classes
- heapq — Heap queue
- bisect — Array bisection
- array — numeric array
- weakref — Weak refs
- types — Dynamic type
- copy — shallow/deep
- pprint — pretty printer
- reprlib — Alt repr()
- enum — enumerations

datetime module

https://docs.python.org/3/library/datetime.html

- Classes:
  - `datetime` , `date` , `time` ; `timedelta`
- Time arithmetic

```
>>> from datetime import *
>>> today = date.today()
>>> now = datetime.now()
>>> tomorrow = today + timedelta(days=1)
>>> nextweek = today + timedelta(weeks=1)
>>> twoweeksago = today - timedelta(weeks=2)
>>> fivehoursfromnow = now + timedelta(hours=5)
>>> threedaysfromnow = today + 3 * timedelta(days=1)
```

  - timedelta doesn't work with years due to leap year

Notes about datetime

- has both class methods and instance methods!
  - `datetime.now()` is a class method
  - today`.replace(`year=2012`)` is an instance method -- without modification to original
- naive vs. aware
  - naive = not aware of timezone
  - aware = know the time zone, daylight saving time

calendar module

https://docs.python.org/3/library/calendar.html

- Purpose: format monthly/yearly calendars
  - text (TextCalendar) or HTML (HTMLCalendar)
  - constructor specifies start DoW: 0=Mon, 6=Sun...

```
>>> import calendar
>>> cal = calendar.TextCalendar(firstweekday=6)
>>> cal.prmonth(2019, 7)
     July 2019
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
>>> cal.formatmonth(2019, 7)
'     July 2019\nSu Mo Tu We Th Fr Sa\n    1  2  3  4  5  6\n 7  8
9 10 11 12 13\n14 15 16 17 18 19 20\n21 22 23 24 25 26 27\n28 29
30 31\n'
```

collections module

https://docs.python.org/3/library/collections.html

`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values
`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

namedtuple class (in collections module)

- subclass of tuple, supports attribute access

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y']) # generate a "class"
>>> p = Point(2, 3) # instantiate named tuple using this "class"
>>> p
Point(x = 2, y = 3)
>>> p.y
3
```

- Still immutable like tuples

```
>>> p.y = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> q = p._replace(y=5) # replace means make a new named tuple
>>> q
Point(x = 2, y = 5)   # new Point, did not modify original
>>> p
Point(x = 2, y = 3)   # old Point q, unmodified
```

Counter class (in collections)

- keeps count for items
  - think: "multiset"
  - implemented as dict { item: count, ... }
- Example

```
>>> from collections import Counter
>>> c = Counter({'red': 4, 'blue': 2}) # can construct from dict
>>> d = Counter('abacadabra')     # also construct from iterable
>>> d
Counter({'a': 5, 'b': 2, 'c': 1, 'd': 1, 'r': 1})
>>> d.most_common(1)
[('a', 5)]
>>> d.most_common(2)
[('a', 5), ('b', 2)]
```

other Counter facts

- allows negative, float, and fraction counts also!
  - initialize or set count value after constructor
- Overloaded operators
  - pairwise +, -, & (min), | (max)
  - unary + to make a new Counter with positive elements
  - unary - to make a new Counter with negative counts, and negate count values to make positive
- least_common = most_common()[-1]

collections.abc module: abstract base classes for containers

https://docs.python.org/3.7/library/collections.abc.html

- purpose:
  - test if a class supports an interface
- Mechanism
  - `isinstance(`*`inst`*`, `*`abc`*`)`
  - `issubclass(`*`cls`*`, `*`abc`*`)`
- even if not actually inherited or instantiated from!

```
>>> import collections.abc as abc
>>> issubclass(list, abc.Container)
True
>>> isinstance('', abc.Generator)
False
```

collections.abc module: abstract base classes for containers

https://docs.python.org/3.7/library/collections.abc.html

- Container
- Hashable
- Iterable
- Iterator
- Reversible
- Generator
- Sized
- Callable
- Collection
- Sequence
- MutableSequence
- ByteString
- Set
- MutableSet
- Mapping
- MutableMapping
- MappingView
- ItemsView
- KeysView
- ValuesView
- Awaitable
- Coroutine
- AsyncIterable
- AsyncIterator
- AsyncGenerator
-

types module

https://docs.python.org/3.7/library/types.html

- way to define new types
- space for standard types not in __builtins__
  - FunctionType
  - LambdaType
  - GeneratorType
  - CoroutineType
  - AsyncGeneratorType
  - CodeType
  - MethodType
  - BuiltinFunctionType
  - BuiltinMethodType
  - WrapperDescriptorType
  - MethodWrapperType
  - MethodDescriptorType
  - ClassMethodDescriptorType
  - ModuleType

examples for type checking

- Can use isinstance() or issubclass() to check instance relationship
  - can also compare class for identity (**`is`**)

```
>>> import types
>>> def gen(): yield 1
...
>>> f = gen()   # make a generator object
>>> isinstance(gen, types.FunctionType)
True
>>> isinstance(gen, types.GeneratorType)
False
>>> isinstance(f, types.FunctionType)
False
>>> isinstance(f, types.GeneratorType)
True
>>> type(f) is types.GeneratorType
True
```

Enumeration generated from Enum class in enum module

- a way to define a group of names for constant values in a domain
  - enumeration members may be ordered or aliased
  - enumeration members may be combinations ("flags")
  - members are instances of the enumeration -- can be tested using isinstance()

Example enumeration

```
class Shake(Enum):
    VANILLA   = 7
    CHOCOLATE = 4
    COOKIES   = 9
    MINT      = 3

>>> for v in Shake:
...     print(v)
...
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
>>>
>>> Shake['MINT']
<Shake.MINT: 3>
```

```
>>> Shake(9)
<Shake.COOKIES: 9>
>>>
>>> Shake.VANILLA > Shake.MINT
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between
instances of 'Shake' and 'Shake'
```

Enum by constructing list

- List syntax, automatic value alignment

```
>>> Animal = Enum('Animal', ['ANT', 'BEE', 'CAT', 'DOG'])
>>> Animal(1)
<Animal.ANT: 1>
>>> Animal(2)
<Animal.BEE: 2>
>>> Animal.CAT
<Animal.CAT: 3>
```

- key-value pairs (tuples) or dict syntax

```
>>> Animal = Enum('Animal', [('ANT', 7), ('BEE', 4), ('CAT', 6),
('DOG', 9)])
>>> Animal(4)
<Animal.BEE: 4>
```

```
>>> Animal = Enum('Animal',{'ANT':7,'BEE':4,'CAT':6,'DOG':9})
>>> Animal(9)
<Animal.DOG: 9>
```

numbers module

- Abstract base classes
  - Number (most general, including complex)
  - Complex (complex: real, imag, conjugate)
  - Real (int, float, but not complex)
  - Rational (has numerator and denominator)
  - Integral (bool, int)

cmath module: complex numbers

https://docs.python.org/3/library/cmath.html

- conversion
  - (x, y) to polar coordinates (abs(c), phase(c))
- functions
  - exp(), log(), log10(), sqrt()
  - acos(), asin(), atan(), cos(), sin(), tan(), acosh(), asinh(), atanh(), cosh(), sinh(), tanh()
- classification
  - isfinite(), isinf(), isnan(), isclose(),
- constants
  - pi, e, tau, inf = float('inf'), infj, nan, nanj,

decimal module, Decimal class

https://docs.python.org/3/library/decimal.html

- Motivation: float is inexact in base-10

```
>>> 1.1 + 2.2           # inexact when represented in binary!
3.3000000000000003
```

- Solution: decimal instead of float
  - works with standard operators

```
>>> from decimal import Decimal
>>> Decimal('1.1') + Decimal('2.2')
Decimal('3.3')
>>> Decimal('3.3').as_integer_ratio()
(33, 10)
```

- Functions: ln(), log10(), ...

fractions module, Fraction class

https://docs.python.org/3/library/fractions.html

- representation of rational numbers
  - i.e., ratio of two integers
  - performs simplification of fraction!

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
```

```
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
```

random module

https://docs.python.org/3/library/random.html

- Generator
  - int: `random.randint(a, b)` inclusive of [a, b]
    `random.randrange(a, b, step)`
  - float: `random.random()` in [0.0, 1.0)
    `random.uniform(a, b)` in [a, b)
- On sequence (e.g., list)
  - `random.shuffle(L)`, `random.choice(L)`, `random.sample(L, k)`
- concept: random seed
  - `random.seed(n)`, defaults to current time
  - same seed => same sequence of random numbers

random module cont'd

```
>>> import random
>>> random.random()               # Random float:  0.0 <= x < 1.0
0.37444887175646646
>>> random.uniform(2.5, 10.0)     # Random float:  2.5 <= x < 10.0
3.1800146073117523
>>> random.randrange(10)          # Integer from 0 to 9 inclusive
7
>>> random.randrange(0, 101, 2) # Even integer from 0 to 100 inclusive
26
>>> random.choice(['win', 'lose', 'draw'])  # pick 1 random element
'draw'
>>> L = [10, 20, 30, 40, 50]
>>> random.sample(L, k=4)         # 4 samples w/out replacement
[40, 10, 50, 30]
>>> random.shuffle(L)
>>> L
[40, 10, 30, 20, 50]
```

random module cont'd

- can instantiate random-number generators
  - why? allow independent operation

```
>>> import random
>>> r = random.Random()  # instantiate a random number generator obj
>>> s = random.Random()
>>> [r.randint(1, 100) for i in range(10)]
[93, 8, 27, 29, 96, 39, 33, 50, 93, 9]
>>> [s.randint(1, 100) for i in range(10)] # different sequence
[6, 50, 59, 5, 42, 98, 14, 5, 90, 4]
>>> r.seed(100)           # set to known seed
>>> [r.randint(1, 100) for i in range(10)]
[19, 59, 59, 99, 23, 91, 51, 94, 45, 56]
>>> s.seed(100)           # same seed => same sequence!!
>>> [s.randint(1, 100) for i in range(10)]
[19, 59, 59, 99, 23, 91, 51, 94, 45, 56]
>>> s.seed(295)           # different seed => different sequence
>>> [s.randint(1, 100) for i in range(10)]
[37, 91, 61, 41, 76, 86, 62, 10, 34, 31]
```

statistics module

- supported functions

`mean()`	Arithmetic mean (“average”) of data.
`harmonic_mean()`	Harmonic mean of data.
`median()`	Median (middle value) of data.
`median_low()`	Low median of data.
`median_high()`	High median of data.
`median_grouped()`	Median, or 50th percentile, of grouped data.
`mode()`	Mode (most common value) of discrete data.

functional programming modules

- itertools
  - additional useful iterators
- functools
- operator

itertools module (1/3)

https://docs.python.org/3/library/itertools.html

- infinite iterators
  - count(start, [step])
  - cycle(iterable)
  - repeat(element [, n])

```
>>> import itertools
>>> c = itertools.count(10)
>>> [next(c) for i in range(10)]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> cy = itertools.cycle('ABC'))
>>> [next(cy) for i in range(10)]
['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C', 'A']
>>> list(itertools.repeat('z', 10))
['z', 'z', 'z', 'z', 'z', 'z', 'z', 'z', 'z', 'z']
```

itertools module (2/4)

- Iterators terminating on the shortest input sequence:
  - accumulate()
  - chain()
  - chain.from_iterable()
  - compress()
  - dropwhile()
  - filterfalse()
  - groupby()
  - islice()
  - starmap()
  - takewhile()
  - tee()
  - zip_longest()

```
>>> from itertools import *
>>> list(accumulate([1, 7, 3, 4, 6]))
[1, 8, 11, 15, 21]
>>> list(dropwhile(lambda x: x<5, [1,4,6,4,1]))
[6, 4, 1]
>>> list(takewhile(lambda x: x<5, [1,4,6,4,1]))
[1, 4]
>>> list(itertools.zip_longest('ABCD', 'WXY',
'12', fillvalue='-'))
[('A', 'W', '1'), ('B', 'X', '2'), ('C', 'Y',
'-'), ('D', '-', '-')]
```

itertools module (3/4)

- Combinatoric Iterators:
  - product()
  - permutations()
  - combinations(), combinations_with_replacement()

```
>>> import itertools import *
>>> list(itertools.product('012', 'abc'))
[('0', 'a'), ('0', 'b'), ('0', 'c'), ('1', 'a'), ('1', 'b'), ('1',
'c'), ('2', 'a'), ('2', 'b'), ('2', 'c')]
>>> list(itertools.permutations('ABC')) # ways of rearranging
[('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C',
'A'), ('C', 'A', 'B'), ('C', 'B', 'A')]
>>> list(itertools.combinations('ABCD', 3)) # pick 3 out of 4 items
[('A', 'B', 'C'), ('A', 'B', 'D'), ('A', 'C', 'D'), ('B', 'C',
'D')]
```

operator module

https://docs.python.org/3/library/operator.html

- Provides function syntax to operators

Syntax	Function
a + b	add(a, b)
seq1 + seq2	concat(seq1, seq2)
obj in seq	contains(seq, obj)
a / b	truediv(a, b)
a // b	floordiv(a, b)
a & b	and_(a, b)
a ^ b	xor(a, b)
~ a	invert(a)
a | b	or_(a, b)
a ** b	pow(a, b)
a is b	is_(a, b)
a is not b	is_not(a, b)
obj[k] = v	setitem(obj, k, v)
del obj[k]	delitem(obj, k)
obj[k]	getitem(obj, k)
a << b	lshift(a, b)
a % b	mod(a, b)

Syntax	Function
a * b	mul(a, b)
a @ b	matmul(a, b)
- a	neg(a)
not a	not_(a)
+ a	pos(a)
a >> b	rshift(a, b)
seq[i:j] = v	setitem(seq, slice(i, j), v)
del seq[i:j]	delitem(seq, slice(i, j))
seq[i:j]	getitem(seq, slice(i, j))
s % obj	mod(s, obj)
a - b	sub(a, b)
obj	truth(obj)
a < b	lt(a, b)
a <= b	le(a, b)
a == b	eq(a, b)
a != b	ne(a, b)
a >= b	ge(a, b)
a > b	gt(a, b)

Standard modules

https://docs.python.org/3/library/

- Data types
- Numeric
  - math, numbers, cmath, decimal, fractions, random, statistics
- Functional Programming
  - itertools, functools, operator
- Text Processing
- File access, data persistence, compression, format, cryptography
- OS service, concurrent execution
- Networking, HTML, XML
- Multimedia, internationalization
- Framework for graphic, GUI, command-line

Standard modules

- Debugging
  - pdb
- math related
  - math, random
- Date and time
  - datetime, calendar
- Data struct, functional programming
  - collections, functools
- Data representation
  - pickle, csv, json, xml,
- User interface
  - tkinter
- string processing
  - string, re (regular expression)
- System programming
  - system, os, os.path, socket
- C-language interface
  - ctypes, decimal

Module vs. Package

- Module by default means a single python file
  - but what if it gets too big or too complex?
  - Solution: package
- Package = "directory" of several modules
  - a package is also a module; can contain other subpackages
  - must contain an `__init__.py` file!
    - defines what other modules in that directory should be imported into the package as a module
    - define `__all__` = list of modules to import