

# PYTHON

Advanced course

Jos de Bruin (jos@iaaa.nl)

ABN AMRO, October 2019

1

## Topics

1. Quick recap of basic elements of Python – the language
  2. Classes
  3. Descriptors / attribute lookup
  4. Decorators
  5. Metaclasses
  6. Iterators, comprehensions, generators
  7. Context managers
  8. Concurrency (threads, multiprocessing, coroutines, asyncio)
  9. Persistence (DB API, XML, JSON, pickle ...)
  10. Modules, packages, installation and distribution, virtual environments
  11. Logging, testing, debugging
  12. Networking, standard library modules
- } Patterns
- } Metaprogramming

2

## Resources

### books

- Python in a nutshell, *Alex Martelli, Anna Ravenscroft, and Steve Holden*, 2017, O'Reilly
- Effective Python, *Brett Slatkin*, 2015, Addison Wesley
- Python Cookbook, *David Beazley and Brian K. Jones*, 2013, O'Reilly
- Python Enhancement Proposals (PEP's): <https://www.python.org/dev/peps/>

questions: [python.org](https://python.org), [stackoverflow.com](https://stackoverflow.com), your favorite search engine

tutorials galore, a suggestion: [David Beazley](#)

style guides: [PEP8](#), <https://google.github.io/styleguide/pyguide.html>

3

## Tools

- Anaconda distribution
- Python 3.7
- Spyder (alternatives: PyCharm, Eclipse+PyDev, SublimeText, Atom, Visual Studio Code ...)
- CLI: IPython or plain Python
- (maybe) Jupyter notebooks

4

## The basics

5

### Python

- a language
- an interpretation (data model)
- an interpreter
  - to run scripts/programs
  - to interact with user: a REPL (read-exec-print-loop)
- an ecosystem (distributions, libraries etc.)
- parsers, compilers

6

## The interpreter

- interpreters for most platforms, and in various languages, C, Java (JPython), .NET (IronPython) ...
- the CPython interpreter, is the reference implementation
- CPython provides for extensions written in C (e.g. numpy)
- tools that make it fairly easy to speed things up by converting modules to C
- current version 3.7, 3.8 in pre-release (2.7 is still used a lot, but will be phased out)

7

## The language

- keywords: predefined symbols (terms, words) with fixed meaning
- lexical rules for the introduction of additional symbols (names/variables)
- how to combine these symbols into acceptable expressions (syntax)
- the data or object model (semantics)
  - the different types of data that Python distinguishes
  - ways to extend the data model (introduce new types)
- how to interpret statements

8

## Language: lexical structure

low-level syntax, rules for grammatically correct program:

- **keywords:** `and continue except global lambda raise yield as def if not return assert del finally import or try break elif for in pass while class else from is with None False True async await`
- **identifiers:** (Unicode) letters, digits and underscore (`_`), but no digit as first character
- **operators:** `+ - * / % ** // << >> & | ^ ~ < <= > >= != ==`
- **delimiters:** `( ) [ ] { } , : . ' = ; @ += -= *= /= //= %= &= |= ^= >= <= **=`
- **literals:**

numbers: `2, 3_000_00, 2.1e3, 4+2j, 0xA3B`  
strings: `'string', 'string', '''multi-line string'''`  
data values of container types:  
`[1,2,3], [], (), ('a',), {}, {1:'a', 'a':2}`

9

## Language: statements

- **simple:**

one per logical line

an isolated expression is a statement, but only useful in REPL or for its side effect ...

<code>a = 3</code>	<code>[1, 2, 3]</code>	<code>a\</code>	<code>= 3; [1, 2, 3]</code>
		<code>= 3</code>	
- **complex:**

consists of multiple statements (clauses), controls their execution

all clauses at same indentation  
clause: header + body  
header: starts with a keyword, ends with `:`  
body (block): one or more statements  
    each statement on its own logical line,  
    all with same indent from header (style guide: 4 spaces)  
in simple statements body can be on same (logical) line as header

10

## The data model

- statements can be expressions (have a value) or instructions (commands)
- an expression always evaluates to some kind of object
- that object itself can be callable (executable)
- `def foo(): pass`
  - `foo` now acts like a variable with a callable as value
  - `foo()` means: call (run) the value
- so data model actually also covers programs: `program = data`

11

## Data types

A data type determines

- its meaning, i.e. how to interpret the bits used to represent a piece of data of that type
- possible values for item
- operations or functions that can be applied to (are supported by) that item

Python provides

- a number of built-in types (*number, string, list* etc.),
- the option to define new types (using a *class* declaration) or to modify existing types

Made possible by using a single, uniform representation for all of its data: the **object**

12

## Everything is an object

Each object stores

- a unique **identifier** (an integer), returned by `id(x)`; in CPython: its memory address
- a **type**, returned by `type(x)`
- a **value**, such as a number or a container of other objects (**items**)
- **attributes** that may store both data and/or callables

An object's type determines its mutability

It *may* have one or more names; these are not stored with the object, but in separate namespaces, i.e. dictionaries `{name: object, ...}`

→ Names do not have types, they can be freely reassigned

13

## Attributes

Each object has

- a number of special attributes, “dunders”, “double\_underscore” attributes
- these are always (meant to be) class or type attributes
- have special meaning and are basis for polymorphism / dispatching of operators
- can be overridden by user-defined types (classes)
- in addition most objects can be given any number of additional attributes, directly or through their type/class

14

## Numbers

- *int*, *float*, *complex*    1\_234\_456, 3.8e3, 3.8+3j, 11 0b01011 0o13 0xB
- *bool* is subtype of *int*,  
values 0 or 1 (printed as: False or True)
- standard library also has:
  - *fractions*
  - *decimal*    (if you want to be sure that 1.1 + 2.2 == 3.3)

numbers are always immutable

if two numbers are equal (`x == y`), they (probably) aren't identical (`x is y`)

15

## Containers

Sequence (bounded iterable)

- mutable: *list*, *bytearray*
- immutable: *tuple*, *string*, *bytes*

Set types

- mutable: *set*
- immutable: *frozenset*

Mapping: *dict*(ionary)

16



## Sequences

- concatenate/repeat:    `+` and `*`
- membership test:      `<exp> in <sequence>`
- indexing:              `sequence[n]`
- slicing:                `seq[start:stop:step]`
- count:                 `len(seq)`

17

## Sequences

- concatenate/repeat:    `+` (`__add__`) and `*` (`__mult__`)
- membership test:      `<exp> in <sequence>`
- indexing:              `sequence[n]` or `sequence.__getitem__(n)`
- indexing:              `seq[start:stop:step]` or  
`seq[slice(start,stop,step)]` or  
`sequence.__getitem__(slice(start,stop,step))`

18

## Dictionaries

- any hashable object can act as key, value can be any object
- `d.keys()`, `d.values()`, `d.items()`: iterables that stay in sync with dict
- insertion order is maintained (>3.7)
- `d.pop`, `popitem`
- `setdefault(k, default)`
- `d.update(other_dict)`
- `collections` module has many other useful container types
- `sets` are basically empty dictionaries: just keys(hashable objects)

19

## bytes and strings

- strings are sequences of unicode code points (total number: `sys.maxunicode = 1114111`)
- need encoding to convert these into bytes, and to convert bytes back to unicode
- most used (e.g. default for scripts): `utf-8`

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

20

## bytes and strings

string is immutable, avoid concatenation, use `s.join(seq)`

all strings are Unicode

`s = "Αποκωδικοποίηση και κωδικοποίηση ελληνικού κειμένου "`

`'{π or by name: \N{GREEK SMALL LETTER PI} or by number: \u03c0}'`

to exchange: conversion to and from bytes

`b = bytes(s, encoding='utf-8')`

`b = s.encode(encoding='utf-8')`

`s == b.decode(encoding='utf-8')`

21

## strings

- immutable
- so concatenation means completely new string
- use `join`
- or `io.StringIO`
- modules `string` and `re` for manipulating and search

22

## Format

```
'{:c}'.format(ord(s[0]))
```

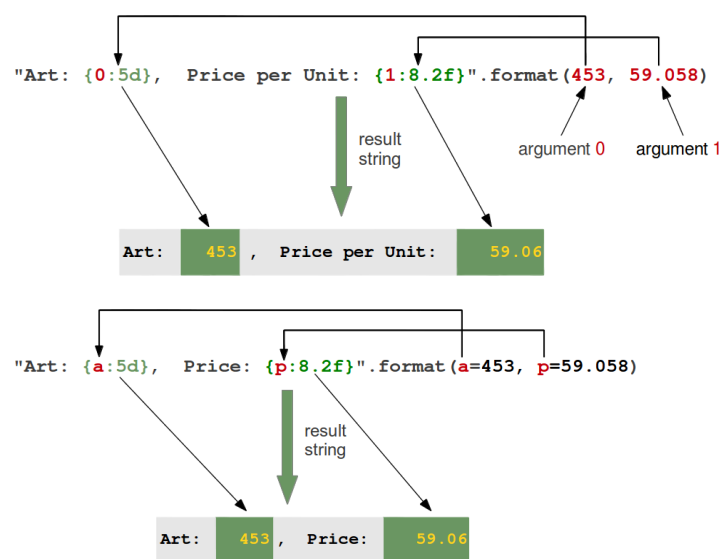
```
f'formatted:{s[1:22]}'
```

other prefixes:

r: raw string

b: bytes

u: unicode



23

## None

None is an object, only instance of `NoneType`

`a = None` means both `a == None` and `a is None` are `True`

use `is` operator (as `__eq__()` can be anything ....)

```
class Strange:
    def __eq__(self, other):
        return True
```

```
funny = Strange()
```

```
funny is None → False
```

```
funny == None → True
```

`None` evaluates to falsity, but beware: `None != False` (same for `[]`, `{}`, `''` etc.).

Other single-valued types:

`NotImplemented` (`NotImplementedType`)

`Ellipsis` (`literal: ...`)

24

## Variables (assignment)

Only way to introduce variables (no notion of defining a variable!)

Basic: `target = expression` (expression is evaluated and bound to target)

target: an identifier, attribute reference, indexing or slicing

assignment to identifier always succeeds, others may raise exceptions

also allowed: `target1 = target2 = target3 = exp`

Augmented assignments: `var *= exp`  $\rightarrow$  `var = var*exp`

25

## Value unpacking

`a, b, c = x` equivalent to: `a=x[0], b=x[1], c=x[2]`

Note: LHS is a sequence (here a tuple, can also be list)

`a, *b, c = [1,2,3,4,5]`  $\rightarrow$  `a==1, b==[2,3,4], c==5`

`*a, b, c = [1,2,3,4,5]`  $\rightarrow$  `a==[1,2,3], b==4, c==5`

`a, b, *c = [1,2,3,4,5]`  $\rightarrow$  `a==1, b==2, c==[3,4,5]`

New in 3.8: assignment expressions:

`a=(b:=2)+(c:=9*b)`  $\rightarrow$  `a==20, b==2, c==18`

26

## Callables

- any object that can be called (i.e. contains or is code that can be run)
- i.e. anything that has an attribute `__call__` (duck typing!)
- functions, generators, methods, instances of callable classes
- called by `()` operator: `callable_obj(*args, **kwargs)`  
`callable_obj.__call__ (*args, **kwargs)`

27

## Functions

```
def identifier(<positional args>[,<named/optional args>]):  
    statement(s)
```

*if first statement is string literal that string becomes value of `__doc__` special attribute*

- optional/default args are specified as assignment:  
`arg1=default, arg2=default`
- defaults are evaluated at definition (so `arg=[ ]` is likely to be bad idea)
- functions returns `None` unless some `return exp` statement is executed

28

## Functions are called by *object reference*

- args are evaluated, resulting objects bound to corresponding function parameters
- these objects are the real thing, not copies, so mutable ones can be mutated by function!
- all parameters can be set by name (par = exp), but positional settings have to come first
- `def foo(a, b, c=3): ...` can be called as:  
`foo(1,2), foo(9,5,3), foo(b=2, a=3),`  
`foo(4, b=2, c=6), foo(*(1,2))`

29

## Variable (optional) arguments

- leaving the exact number of arguments open:
 

```
def foo(*args, **kwargs):
    type(args) is tuple
    type(kwargs) is dict
```
- option: argument settings in dict: call `foo(**dict)`
- or: argument settings in iterable, call `foo(*iterable)`
- to force use of named parameters:
 

```
def foo(a, *, b, c=3):
```
- to force positional only:  
 [in 3.8, but already used in help]
 

```
def foo(a, /, b, c):
```

30

## Scope / namespace

- *global variables* are attributes of the module object and make up the global namespace: attributes of module object
- the *local scope* or *namespace* of a function: its parameters plus names bound in its body
- *globals* have a value in local scopes if not shadowed (hidden) by similarly-named locals
- but to set them, they have to be declared `global` in the local scope (function body) – universally frowned upon

31

## Lexical scope

- nested functions can use names declared in their “nesting” function(s)
- such *free variables* can only be reassigned if declared `nonlocal`
- `foo` returns a **closure** of `baz`

```
def foo(x):
    def baz(n):
        return n + x
    return baz
```

`foo(10)(3)` returns 13

```
def foo(x):
    def baz(n):
        nonlocal x
        x += n
        return x
    return baz
```

Note: a class also defines a namespace

32



## Lambda expression

```
lambda parameters: expression
filter (lambda x: x+1 if x % 2 else x, range(10))
nameless, but still a function object
```

33

## Control

branching:

```
if condition:
    statement(s)
elif condition:
    pass
elif ...
else:
    statement(s)
```

```
with open('filepath') as f:
    for line in f:
        pass
```

iteration:

```
for x in iterable:
    statement(s)
else:
    statement(s)
```

or

```
while condition:
    statement(s)
else:
    statement(s)
```

```
if condition:
    statement(s)
elif condition:
    break
elif condition:
    continue
else:
    statement(s)
statement(s)
```

34

## else

keyword for optional last clause in *loop*, *while* or *try* statement

A confusing name, but can be useful.

*Do this if things proceed normally, i.e.:*

- for runs to completion (no break),
- while finishes because condition becomes *falsy* (no break),
- no exception was raised in try block

e.g. when looking for something:

```
for item in my_list:
    if searched_for(item):
        break
else:
    raise ValueError('Nothing found!')
```

restrict try block to statements try is meant to guard

```
try:
    tricky_stuff()
    other_stuff()
except Exception:
    cleanup()

try:
    tricky_stuff()
except Exception:
    cleanup()
else:
    other_stuff()
```

35

## Exceptions

to raise exception:

```
raise Exception(args)
```

raise

without argument in except block  
re-raises current exception (perhaps  
after some editing)

lot's of predefined exceptions, but  
can always add your own type

```
try:
    tricky_stuff()
except ExceptionA as e:
    handle(e)
except ExceptionB as e:
    do_stuff
    raise
except (ExceptionC, ExceptionD) as e:
    raise ExceptionE(args)
except:
    <every error except A, B, C or D>
else:
    other_stuff()
finally:
    wrap_things_up
```

36

get full memorysize

```
import sys
def get_size(obj, seen=None):
    """Recursively finds size of objects"""
    size = sys.getsizeof(obj)
    if seen is None: seen = set()
    obj_id = id(obj)
    if obj_id in seen: return 0
    # Mark as seen *before* entering recursion to handle self-referential objects
    seen.add(obj_id)
    if isinstance(obj, dict):
        size += sum([get_size(v, seen) for v in obj.values()])
        size += sum([get_size(k, seen) for k in obj.keys()])
    elif hasattr(obj, '__dict__'): size += get_size(obj.__dict__, seen)
    elif hasattr(obj, '__iter__') and not isinstance(obj, (str, bytes, bytearray)):
        size += sum([get_size(i, seen) for i in obj])
    return size
```

37

## Classes

38

## Classes (aka types)

```
class name(*bases):
    statement(s)
```

the `class` statement is used to define new types:

- creates class object
- stores bases on it (bases are other types from which this class inherits attributes)
- executes statements (different from function definition!)
- if first statement is literal string, stores that string in `__doc__` of class object to retrieve: `help(class)`
- assignments and function definitions in body are stored in `__dict__` of class object (all of these are called *attributes*, whether bound to a method or other type of object)

style: names are in CamelCase (except when built-in, written in C)

39

## Classes

- use `__init__(self, *a, **kw)` for initialization of an instance
- note: `__init__` can only change attributes, not return other object (or anything else)
- the object is created by special method `__new__`
- `Cls(*args, **kwargs)` ↔

```
class Foo:
    def __init__(self, x):
        self.x = x
    def baz(self):
        print(self.x)
```

```
x = Cls.__new__(Cls, *args, **kwargs)
Cls.__init__(x, *args, **kwargs)
return x
```

40

## Example

```
class Employee:
    'Common base class for all employees'    # doc, returned by: help(Employee)
    empCount = 0                             # class attributes
    def __init__(self, name, salary):         # called when object is created
        self.name = name                    # attributes set on new object
        self.salary = salary
        Employee.empCount += 1              # class attribute updated
    def __str__(self):
        print("Name: ", self.name, " , Salary: ", self.salary)

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

41

## Subclassing

```
class Employee:                                # a base class
    def __init__(self, name): self.name = name # initialized with name
    def create_badge(self): return "{} works here".format(self.name)
class WageEmployee(Employee):                  # define derived class
    def __init__(self, name, rate):
        super().__init__(name)
        self.rate = rate
        self.hours = 0
    def worked(self, hours): self.hours += hours
    def pay_check(self): "{}: wage {}".format(self.name, self.hours*self.rate)
```

42

## Class: a callable that returns a new instance of that class

```
e = Employee('Joe'); w = WageEmployee('Jane', 20)
```

- attributes on its class act like *virtual* attributes of the instance, in addition to the attributes stored directly on object: `inst.att` returns a value if `att` is found on object, its class or one of its base classes, in that order

```
w.pay_check(); e.create_badge(); w.create_badge(); e.pay_check()
```

- functions defined within `class` statement are called instance methods, because they are meant to be called with objects of that class as their first argument:

43

## instance methods

- instance methods need access to the specific instance they are called on
- in Python this is done by explicitly passing in the instance as argument to method
- `foo.method(*args, **kwargs) == Foo.method(foo, *args, **kwargs)`
- so you need to add a parameter (in front) to its signature in its `def` statement
- convention: name that parameter (i.e. the instance) **self**

```
class Foo:
    def baz(self):
        print(self.x)
foo = Foo()
foo.baz()
```

44

## Class: overriding attributes, including the special ones

- a class can always override attributes inherited from its bases
- all classes derive from object (i.e. have object as a base by default)
- special methods (“dunders” because they start and end with double underscores) are inherited from object or other built-in types and can be overridden if objects of this class should be handled differently by the corresponding operators or builtin functions;
  - `__str__` method is called by `str(<some_employee>)`
  - `__init__` is automatically called after new object is created

45

## Attribute access

attributes can be accessed and added, removed or modified, using dot notation:

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

Alternative for dealing with 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
```

```
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'
```

46

## Super

- to call instance method of some other class Cls:
  - call it directly: `Cls.att(instance, *args, **kwargs)`, or
  - call it on instance of super class: `super(BaseClass, obj)`
- a super instance holds reference to obj and to BaseClass; it is a descriptor, “redirecting” an attribute lookup to the BaseClass and inserting obj instead of itself in the call of the instance method found
- used inside a method definition, `super()` i.e. without args, is taken to mean `super(<current_class>, <instance on which method is being called - the self arg>)`

```
class B:
    def foo(self):
        pass
class C:
    def func(self):
        pass
class A(B):
    def foo(self):
        super().foo()
    def baz(self):
        super().foo()
        C.func(self)
```

47

## Inheritance

- with multiple bases, inheritance is not over trees, but over directed graphs
- super needs a linearization, a fixed order in which to check the bases: this order is called the Method Resolution Order
- fixed at class declaration, stored under `__mro__` attribute
- fails when no order can be found that is both monotonic (a subclass does not force different order than its ancestors) and reflects the bases ordering in each of the ancestors
- multiple inheritance from built-in types (generally) not allowed: `TypeError`

```
class X: pass
class Y: pass
class A(X,Y): pass
class B(Y,X): pass
class Z(A,B): pass
```

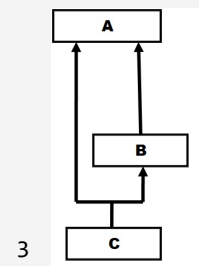
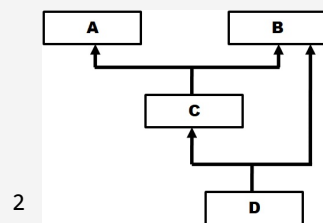
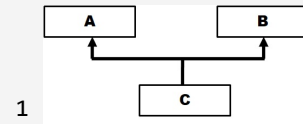
`TypeError: no consistent method resolution order (MRO) for bases X,Y`

48



## Inheritance

1.  $MRO(C) = (A, B)$
2.  $MRO(D) = (C, A, B)$ , but fails if bases for class D are switched from (C,B) to (B, C)
3. Fails, succeeds if bases for C would be (B,A)



49

## 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()
will print:
Constructor A was called
Constructor B was called
Constructor C was called
  
```

In general, certainly with `__init__`, the "baser" initialization should occur before the more specific one.

50

## Destroying objects

- Python periodically reclaims memory by deleting unreachable objects: garbage collection
- Python's garbage collector runs automatically during program execution
- objects deleted when their reference count reaches zero
- reference count changes as references to it change:
  - plus one when it's assigned a new name or placed in a container (list, tuple, or dictionary)
  - minus one when one of its names is deleted, reassigned or goes out of scope, or when it is removed from collection, using `del`
- `__del__()` method:
  - called by system when object is actually destroyed
  - can be used to clean up any non-memory resources used by an instance

51

## Class methods

```
class Bank:
    @classmethod
    def is_valid(cls, key): # code for determining validity here
        <some test specific cls>
    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

52

## Static methods

Static methods do not receive `self` or `cls` as first parameter :

- regular function but has to be called in namespace of class

Method can be made static in two ways :

1. with `@staticmethod` decoration
2. explicitly rebinding name to result of `staticmethod` with `method` as parameter

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

53

## Attribute lookup: descriptors

54

## Attribute lookup: evaluating `obj.attr`

- special attributes (“dunders”) are stored directly on (class) object
- regular attributes are stored in dictionaries (under `__dict__`)
- lookup att on obj: `obj.__getattr__('att')`
- objects found at attribute can be called instead of simple being returned
- Descriptor protocol handles this:
 

```
descr.__get__(self, obj, type=None) -> value
descr.__set__(self, obj, value) -> None
descr.__delete__(self, obj) -> None
```
- if only `__get__`: non-overriding descriptor; if `__set__`: an overriding (or data) descriptor

55

## Attribute lookup on classes:

```
def __getattr__(cls, att): # self is a class
    if att not in cls.__dict__:
        "move up to next class on mro and try again"
    else v = cls.__dict__[att]
    if hasattr(v, '__get__'):
        return v.__get__(None, cls)
    else return v
```

56

### Attribute lookup on instances:

```
def __getattribute__(x, att): # x is an instance
    if type(x).__dict__[att] is an non-overriding descriptor v:
        return v.__get__(x, None)
    elif att in x.__dict__:
        return x.__dict__[att]
    else
        "do the class lookup as shown earlier"

if this lookup fails:
    try special method x.__getattr__(att), which should raise AttributeError
    same error if no __getattr__ defined (default)
```

has a `__get__` method

57

### Attribute setting:

```
on class C:
    C.__dict__[att]=val

on instance x:
    if x has __setattr__ (somewhere on MRO), call that
    elif type(x).att is overriding descriptor (i.e. has __set__ method)
        type(x).__dict__[att].__set__(x, val)
    else
        x.__dict__[att] = val
```

58

## \_\_slots\_\_

```
class C:
    __slots__ = ('a', 'b')
```

- objects will not get `__dict__`, saves memory (values are simply stored in list on object)
- can only assign values to these slots
- uses descriptor protocol to handle getting and setting
- setting the slot attributes on class destroys the descriptor, so that value becomes value for all objects

59

## Bound methods

- functions are non-binding descriptors (they have a `__get__`)
- when accessed as an attribute on an instance, their `__get__` method get called
- this wraps the function in a bound method:

```
class Function():
    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        if obj is None:
            return self
        return method(self, obj)
```

- calling a method `m` is like calling `m.__func__(m.__self__, *args, *kwargs)`

60

## Static and class methods

- *class method*: function defined in class namespace with first argument bound to class (instead of to instance)
- *static method*: regular function defined in class namespace

```
class classmethod:
    def __init__(self, f):
        self.f = f
    def __get__(self, obj, cls=None):
        if cls is None:
            cls = type(obj)
        def newfunc(*args):
            return self.f(cls, *args)
        return newfunc
```

```
class C:
    def foo(...):
        body
    def baz(cls, ...):
        body
    baz = classmethod(baz)
    foo = staticmethod(foo)
```

```
class staticmethod:
    def __init__(self, f):
        self.f = f
    def __get__(self, obj, cls=None):
        return self.f
```

61

## Properties

- built-in overriding descriptor type, to turn attributes into *properties*
- control over setting or reading of attributes, without affecting user interaction
- either: `attrib = property(fget=None, fset=None, fdel=None, doc=None)`

```
class C:
    def __init__(self):
        self._x = None
    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "About x.")
```

62

## Per-instance methods

An instance can have instance-specific bindings for all attributes, including callable attributes (methods). Except for (class) attributes bound to overriding descriptors, instance-specific bindings hide class-level bindings and return the (callable) object bound to the instance.

Note: special methods called implicitly for operations always refer to the class-level bindings:

```
def fake_get_item(self, idx): return idx
class MyClass(object): pass
n = MyClass()
n.__getitem__ = fake_get_item
print(n[23]) → TypeError: unindexable object
```

63

## Decorators

64



## 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
```

65

## Python supports functional programming style

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!
```

Function can take other function as an argument:

```
def call_func(func, *args):
    return func(*args)
```

66

## Closure

Closure is combination of code and scope:

- Functions combine code to be executed and scope in which to do that
- Variables created in outer scope remain readable even after that scope ceases to exist (popped from stack)

```
def startAt(start):
    def incrementBy(inc):
        return start + inc
    return incrementBy
f = startAt(10)
g = startAt(100)
print f(1), g(1)  # print 11 101

def compose_greet_func(name):
    def get_message():
        return "Hello "+name+"!"
    return get_message
greet = compose_greet_func("Joe")
print (greet()) => Hello Joe
```

67

## Decorator syntax

decorator refers to

1. a decorating function
2. syntactic sugar for handling common pattern:  
wrap a function and rebind its name to the wrapper, as in `foo = decorator(foo)`

```
def decorator (decorated):
    def wrapper():
        ...
        decorated ()
        ...
    return wrapper

def foo():
    pass

foo = decorator(foo)

@decorator
def foo():
    pass
```

68

## Decorators

```
x = property(getx, setx, delx, "About x.")
```



```
class C:
    def foo(...):
        body
    def baz(cls, ...):
        body
    baz = classmethod(baz)
    foo = staticmethod(foo)
```



```
class C:
    @staticmethod
    def foo(...):
        body
    @classmethod
    def baz(cls, ...):
        body
```

```
class C:
    def __init__(self):
        self._x = None
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        self._x = value
    @x.deleter
    def x(self):
        del self._x
```

69

## Multiple decorators

decorators can be stacked,  
like any expression evaluated  
inside-out, i.e. bottom-to-top

```
@div_decorate
@p_decorate
@strong_decorate
def get_text(name):
    return "Hi {0}!".format(name)
print (get_text("Mary"))
```

```
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
    "<strong>{0}</strong>".format(func(name))
    return func_wrapper

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

70

## Wrappers

- a wrapper (also called adapter) is an extra layer between caller and function
- it keeps the signature and adds statements, before or after function, or changes the signature, for instance by binding some of the free variables (→ closure)
- NOTE: to be generally useful wrapper should pass on any combination of arguments
- issue: signature of wrapped function is replaced with that of wrapper
- solution: `functools.wraps`
- copies over `__name__, __doc__, __dict__`

```
def wrapper (f):
    def wrapped(*args, **kwargs):
        statements
    return wrapped
```

```
from functools import wraps
def logged(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print(func.__name__ + " called")
        return func(*args, **kwargs)
    return with_logging
```

71

```
def debug(func):
    if 'DEBUG' not in os.environ:
        return func
    msg = func.__qualname__
    @wraps(func)
    def wrapped(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapped
```

## How do we add args to wrapper?

Say we want to add a prefix to debug info?

```
@decorator(args)
def func():
    pass
e.g.
@debug(pref='#')
```

which evaluates to:

```
debug(prefix)(func)
```

`debug` has to become a function that produces another one that does the actual decoration

72

## Decorators with args

debug has to be a function that produces another decorator that does the actual decoration

```
from functools import wraps
def debug(prefix=''):
    def decorator(func):
        msg = prefix + func.__qualname__
        @wraps(func)
        def wrapped(*args, **kwargs):
            print(msg)
            return func(*args, **kwargs)
        return wrapped
    return decorator
```

73

## Decorator with args, alternative solution

```
from functools import wraps, partial
def debug(func=None, *, prefix= ''):
    if func is None:
        return partial(debug, prefix=prefix)
    msg = prefix + func.__qualname__
    @wraps(func)
    def wrapped(*args, **kwargs):
        print(msg)
        return func(*args, **kwargs)
    return wrapped
```

partial returns a function in which some parameters are fixed to the values it was given: a (partial) closure

74

## Class decorators

Not surprisingly, classes can also be decorated

Class decorators

- take a class object and return a wrapper that should return an appropriate class object: the same one, perhaps modified, or even a completely new or different one
- used to give classes extra properties / additional behavior when created
- metaprogramming, can be used for e.g. Factory pattern

75

## class decorator

```
def singleton(cls):
    def wrapped(*args):
        if cls not in instances:
            instances[cls]=cls(*args)
        return instances[cls]
    return wrapped
```

```
@singleton
class Foo: pass
```

Decorators are not inherited, which means duplication of effort; a Singleton class could be option, more general solution is to consider this a characteristic of a metaclass, a whole range of classes that may not have anything else in common ...

76

## Callables with state: three options

### *Closure*

```
def adder_as_closure(augend):
    def add(addend, _augend=augend):
        return addend+_augend
    return add
```

### *Bound method*

```
def adder_as_bound_method(augend):
    class Adder:
        def __init__(self, augend):
            self.augend = augend
        def add(self, addend):
            return addend+self.augend
    return Adder(augend).add
```

### *Callable instance*

```
def adder_as_callable_instance(augend):
    class Adder:
        def __init__(self, augend):
            self.augend = augend
        def __call__(self, addend):
            return addend+self.augend
    return Adder(augend)
```

77

## Metaclasses

78

## Creating a class

`type(object)` → the object's type

`type(name, bases, dict)` → a new type

```
type(5) → int
type(int) → <class 'type'>
type(object) → <class 'type'>
```

```
type.__bases__ → (<class 'object'>,)
type(type) → <class 'type'>
```

```
class Foo:
    a = 0
    def __init__(self, name):
        self.name = name
    def baz(self):
        return "baz"
```

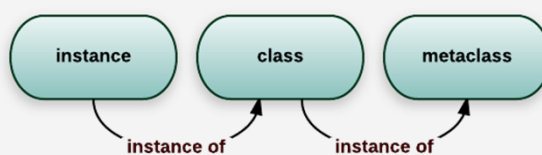
```
def init_Foo(self, name):
    self.name = name
Foo2 = type("Foo",
            (),
            {"a":0,
             "__init__": init_Foo,
             "baz": lambda self: "baz"})
```

`type` acts as a “meta” class creating (instantiating) a class object of type “meta class”

79

## Metaclasses

classes (themselves objects) are built by metaclasses.



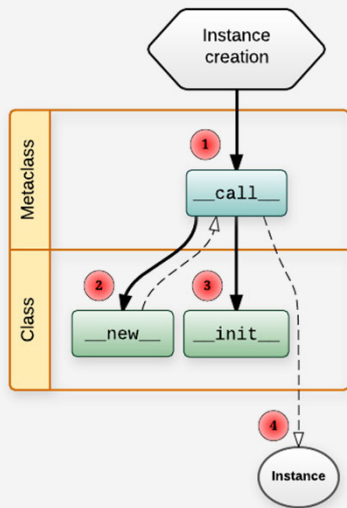
`type` is the default metaclass (the metaclass of type object)

```
class MyType(type):
    pass
class MyClass(metaclass=MyType):
    pass
```

**MyClass created by:**  
`MyType (name, bases, dict)`  
**instead of** `type (name, bases, dict)`

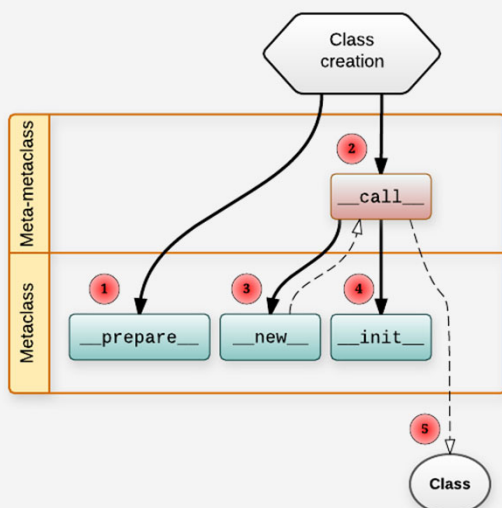
80





1. An instance is created by calling a class object. Its `___call___` method is inherited from its metaclass, i.e. directly or indirectly from `type`.
2. `___call___` calls `___new___` which returns object of appropriate type
3. then `___call___` calls `___init___` on that object
4. and finally returns the initialized object

81



Class creation follows a similar protocol, but with an extra (first) step:

`___prepare___(classname, *classbases, **kwargs)`

`class X(*bases, metaclass=MC, foo='bar')`  
calls:

`MC.__prepare__('X', *bases, foo='bar')`

if present, `___prepare___` must return mapping **m**

- m is used first as namespace for body exec
- then as 3<sup>rd</sup> arg to `MC.__call___`:

`MC.__call__('X', *bases, m)`

82

## Singleton as a type

```
class Singleton(type):
    instance = None
    def __call__(cls, *args, **kwargs):
        if not cls.instance:
            cls.instance = super(Singleton, cls).__call__(*args, **kwargs)
            # or just: cls.instance = super().__call__(*args, **kwargs)
        return cls.instance
class ASingleton(metaclass=Singleton):
    pass
>>> ASingleton() is ASingleton()
>>> True
```

83

## Alternative: a Singleton class

```
class Singleton:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = object.__new__(cls, *args, **kwargs)
        return cls._instance

class Foo(Singleton):
    pass
```

but often more efficient if computation can be done once at metaclass creation

84

## Implementing \_\_slots\_\_

```
class Member:                                # Descriptor implementing slot lookup
    def __init__(self, i):
        self.i = i
    def __get__(self, obj, type=None):
        return obj._slotvalues[self.i]
    def __set__(self, obj, value):
        obj._slotvalues[self.i] = value

class Type(type):                             # Metaclass that implements _slots_
    def __new__(self, name, bases, namespace):
        slots = namespace.get('_slots_')
        if slots:
            for i, slot in (slots):
                namespace[slot] = Member(i)
            orig_init = namespace.get('__init__')
            def __init__(self, *args, **kwargs): # Create _slotvalues
                self._slotvalues = [None] * (slots)
                if orig_init is not None:        # Call orig_init
                    orig_init(self, *args, **kwargs)
            namespace['__init__'] = __init__
        return type.__new__(self, name, bases, namespace)
```

85

## Abstract classes

86

## Abstract base classes (ABCs)

- invocation: call methods, leave it to their implementation to decide on appropriateness and how to handle your specific object (polymorphism)
- inspection: let external code check type or properties of object and use that info to decide on proper way to handle object
- Python is open to inspection, but a more formal, structured way can be desirable: to decide whether an object/class is a Sequence checking for base list is too restricted, for `__getitem__` too broad
- Abstract Base Classes: classes added into inheritance tree to signal certain features to an external inspector

87

## Abstract base classes (ABCs)

- a way to organize tests to determine type
- agree on types, but which: Set, ComposableSet, MutableSet, HashableSet, MutableComposableSet, HashableComposableSet?
- how to fit (keep unaffected) the current type system?
  - e.g. allow Sequence as (virtual) super of e.g. list and tuple
- idea: overload `isinstance(obj, cls)` and `issubclass(sub, cls)`
- add metaclass ABCMeta that:
  - adds way to register subclasses (including builtin types)
  - add abstract methods, which need implementing if class is to be instantiated

88

## Abstract base classes: overloading type checking

```
class ABCMeta(type):
    def __instancecheck__(cls, inst):
        return any(cls.__subclasscheck__(c)
                    for c in {type(inst), inst.__class__})
    def __subclasscheck__(cls, sub):
        candidates = cls.__dict__.get("__subclass__", set()) | {cls}
        return any(c in candidates for c in sub.mro())

class Sequence(metaclass=ABCMeta):
    __subclass__ = {list, tuple}
```

89

## Abstract base classes: registering subclasses

```
from abc import ABCMeta
class MyABC(metaclass=ABCMeta): pass

MyABC.register(tuple)
assert issubclass(tuple, MyABC)
assert isinstance(), MyABC
```

```
from abc import ABC
class MyABC(ABC): pass

MyABC.register(tuple)
assert issubclass(tuple, MyABC)
assert isinstance(), MyABC
```

- `Cls.register(subcls)` is method added by `ABCMeta`, to register subclasses
- `abc.ABC` is convenience: class that just inserts the `ABCMeta` metaclass

90

## Abstract methods

```
from abc import ABCMeta
class MyABC(metaclass=ABCMeta): pass
```

```
MyABC.register(tuple)
assert issubclass(tuple, MyABC)
assert isinstance(), MyABC)
```

```
from abc import ABC
class MyABC(ABC): pass
```

```
MyABC.register(tuple)
assert issubclass(tuple, MyABC)
assert isinstance(), MyABC)
```

- `abc.ABC` is convenience: class that just inserts the `ABCMeta` metaclass

91

## Abstract methods

- `ABCMeta` also supports a new type of method: `abstractmethod`
- can also be used as decorator
- possible (and useful) to mix regular and abstract methods
- even possible to override regular method with abstract method!

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def number_of_wheels(self):
        pass

class Car(Vehicle):
    def number_of_wheels(self):
        return 4

c = Car()

# Try to create a Vehicle: FAILS
v = Vehicle()
```

92

## collections.abc

module providing types for various containers

e.g. `Iterator`:

provides default implementation for `__iter__` and forces you to implement `__next__`

```
from collections.abc import Iterator
import random, math

class Die(Iterator):
    def __next__(self):
        return math.ceil(random.random() * 6)
    def throw(self, n, stop=50):
        assert 0 < n < 7, '1 to 6 please'
        for i in range(1, stop+1):
            d = next(self)
            print(d, end=' ')
            if n == d:
                print('\nThrows: {}'.format(i))
                break
        else:
            print('Looks like a faulty die')
```

93

## Iterables, iterators and generators

94

## Iteration

Python for statement can iterate over many kinds of objects

- over a sequence (items): 

```
for x in [1,4,5,10]: print (x)      # 1 4 5 10
```
- over a dictionary (keys): 

```
for key in {'GOOGLE': 490.10, 'YAHOO': 21.71}:  
    print(key)      # GOOGLE YAHOO
```
- over a string (characters): 

```
for c in "Mars!": print(c)      # M a r s !
```
- over a file gives (lines): 

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

This generic way of handling sequences is not restricted to for statement:

`list(x)`, `tuple(x)`, `max(x)`, `min(x)`, `sum(x)` etc.

all work on many different types of x, as long as they can be treated as Iterable

95

## Iterables and iterators: two protocols

iterable      `iter(iterable)` returns an iterator

iterator      `next(iterator)` returns something

                 or raises an `StopIteration` exception (when exhausted)

`iter(iterator)` returns an iterator

```
for x in c:  
    statement(s)
```

is equivalent to:

```
_iterator = iter(c)  
while True:  
    try: x = next(_iterator)  
    except StopIteration: break  
    statement(s)
```

iterables: list, tuple, set, dictionary, string, but also e.g. file, array, dataframe .....

iterables provide lazy (on-demand) evaluation: e.g. range, enumerate, zip

96



## Iterators

```
_iterator = iter(c)
while True:
    try: x = next(_iterator)
    except StopIteration: break
    statement(s)
```

iterators can only “move forward”,  
cannot be restarted

```
c = Countdown(5)
# or c = countdown(5)
for i in c:
    print(i)
next(c) → StopIteration
```

```
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
```

97

## Comprehensions: functional programming with iterables

if for loop is used to produce new sequence, comprehensions can be useful:

```
l = []
for x in c:
    if test(x):
        l.append(f(x))
```

↔

```
l = [f(x) for x in c if test(x)]
```

Advantages:

- shorter
- expressions instead of statements → can be nested inside other expressions
- the crucial step (applying  $f$  to  $x$ ) is mentioned outside, instead of inside
- var  $x$  is local to comprehension, not in scope of for statement

98

## Generator expressions

`x for x in <iterable>` is a *generator expression*, itself an iterable

`c for w in 'a short sentence'.split() for c in w`

can also add condition at end:

`c for w in 'a short sentence' for c in w if c != ' '`

• `c for w in 'a short sentence'.split() if len(w) > 5 for c in w`

list comprehension: `[c for c in 'a short sentence']`

set comprehension: `{c for c in 'a short sentence'}`

dict comprehension: `{k:v for k,v in enumerate('a short sentence')}`

NOTE: `(x for x in 'a short sentence')` is not a tuple, but an expression

99

## Generator expression

```
ge = (2*x for x in range(5))
print (ge)           # <generator object <genexpr> at ...>
print (next(ge))     # prints 0
for i in ge: print (i) # prints 2, 4, 6 and 8
```

- Generator expressions can be used as arguments to function or in other expression.
- If single function argument, parentheses can be dropped: `sum(x*x for x in s)`

```
sum(2*x for x in range(5)) or sum((2*x for x in range(5))) or sum((ge))
```

- in comprehensions the expression must be without parentheses

```
[2*x for x in range(5)] or [(2*x for x in range(5))] or [*ge]
```

100

## Many functions in Python have become lazy (return iterables)

- `range(5)` returns a range object, an iterable
- `enumerate(iterable)` an `enumerate`: will produce tuples (order, value)
- `zip(*iterables)` a sequence of tuples of length: `len(iterables)`, with the first, second etc. values from each of the iterables
- `map(f, *iterables)` produces results of applying `f` to zip of iterables
- `filter(f, *iterables)` produces results of applying `f` to zip of iterables

```
from operator import add
list(map(add, range(5), range(5,10)))
list(filter(lambda x: x>3, range(9)))
```

101

## Iterator functions

### Built-in:

```
map(func, *iterables)
filter(pred, iterable)
enumerate(iterable, start=0)
zip(*iterables)
```

### in *itertools*:

```
starmap(func, iterable) #unpacks items
accumulate(iterable, [func]) #pairwise
compress(iterable, selector_iter)
chain(*iterables)
chain_from_iterable(it_of_iterables)
zip_longest(*iterables, fillvalue=None)
groupby(clustered_iterable, key=None)
tee(it, n=2)
count(start=0, step==1)
permutations(it, out_len=None)
combinations(it, out_len)
cycle(iterable)
```

102

## Dictionary comprehension

standard dict constructor:

```
d = dict(zip(keys, values))
d = dict.fromkeys('dictionary',0)
```

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

comprehension allows more control:

```
d = {k.upper():v for k,v in zip(keys, values)}
d = {k.upper():0 for k in 'dictionary'}
```

103

## Generators

more convenient way to create iterators

```
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
```

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

↔

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

for i in countdown(5):
    print(i)
```

With a yield statement in its body  
a regular function becomes a  
generator-producing function.

104

## Generators: yield from

more convenient way to yield from nested generators

```
def chain(*iterables):
    for it in iterables:
        for i in it:
            yield i
```

↔

```
def chain(*iterables):
    for it in iterables:
        yield from it
```

- seems minor (syntactic sugar), but: crucial for building **co-routines**
- direct channel from client of co-routine to inner generator

105

## Generators

```
import re
from reprlib import repr

pat = re.compile(r'\w+')

class Sentence:
```

```
    def __init__(self, text):
        self.text = text
        self.words = pat.findall(text)
```

```
    def __repr__(self):
        return 'Sentence(%s)' % repr(self.text)
```

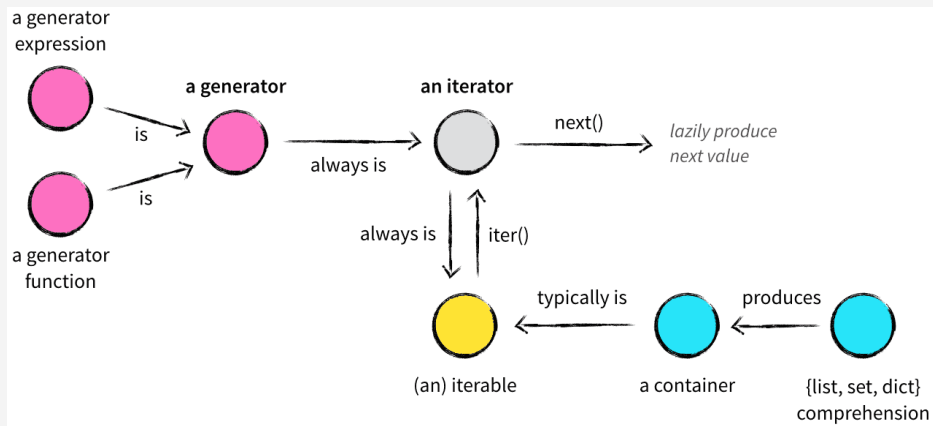
```
    def __iter__(self):
        for word in self.words:
            yield word
        return
```

```
def __init__(self, text):
    self.text = text
```

```
def __iter__(self):
    for match in pat.finditer(self.text):
        yield match.group()
```

106

## Iterables vs. Iterators vs. Generators



107

## Example: Fibonacci

```
def fibonacci(n):
    assert n>0, 'n should be > 0'
    a, b = 0, 1
    while n:
        yield a
        a, b = b, a+b
        n -= 1
```

```
def fibonacci(max):
    assert max>0, 'max should be > 0'
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b
```

```
def fibonacci(n):
    a, b = 0, 1
    while n>0:
        yield a
        a, b = b, a+b
        n -= 1
```

108

## Another example: analyzing log

How many bytes of data were transferred? Or: how to sum last data column in huge 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/ HTTP/1.1" 304 -
```

```
log = open("log")
entries = (line.rsplit(None,1)[1] for line in log)
total = sum(int(x) for x in entries if x != '-')
```

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

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

109

## Context managers

110

## Context managers

The problem: change context to execute code, and reset context when code exits, whatever happened:

*set things up*

try:

*do something*

[except: ...]

finally:

*tear things down*

try:

fd = open('path')

statements

finally:

fd.close()

with *manager* as *handle*:

*statements*

manager object provides the methods

`__enter__` and `__exit__`

```
with open('filepath') as f:
    for line in f:
        pass
```

111

## Context managers

```
with controlled_execution() [as handle]:
    statement(s)
```

```
class controlled_execution:
```

```
    def __enter__(self):
```

```
        set things up
```

```
        return thing
```

```
    def __exit__(self, exc, val, traceback):
```

```
        tear things down
```

```
class File():
```

```
    def __init__(self, filename, mode):
```

```
        self.file = filename
```

```
        self.mode = mode
```

```
    def __enter__(self):
```

```
        self.open_file = open(self.file,
                               self.mode)
```

```
        return self.open_file
```

```
    def __exit__(self, *args):
```

```
        self.open_file.close()
```

```
files = []
```

```
for _ in range(10000):
```

```
    with File('foo.txt', 'w') as infile:
```

```
        files.append(infile)
```

112



## Semantics

```
with expression [as varname]:
    statement(s)
```

```
_normal_exit = True
_manager = expression
varname = _manager.__enter__()
try:
    statement(s)
except:
    _normal_exit = False
    if not _manager.__exit__(*sys.exc_info()):
        raise
# exception does not propagate if __exit__ returns a true value
finally:
    if _normal_exit:
        _manager.__exit__(None, None, None)
```

113

## Managing the context:

```
mgr = (EXPR)
exit = type(mgr).__exit__ # Not calling it yet
value = type(mgr).__enter__(mgr)
exc = True
try:
    try:
        VAR = value # Only if "as VAR" is present
        BLOCK
    except: # The exceptional case is handled here
        exc = False
        if not exit(mgr, *sys.exc_info()):
            raise # The exception is swallowed if exit() returns true
finally: # Normal and non-local-goto cases handled here
    if exc:
        exit(mgr, None, None, None)
```

114

## Context managers

can be nested:

```
with A() as a, B() as b:  
    statement(s)
```

→

```
with A() as a:  
    with B() as b:  
        statement(s)
```

defined by decorating a generator:

```
from contextlib import contextmanager  
  
@contextmanager  
def open_file(name):  
    f = open(name, 'w')  
    yield f  
    f.close()  
  
with open_file('some_file') as f:
```

115

## Concurrency

116

## Concurrency

- *multi-tasking*: performing tasks “simultaneously”, to keep multiple customers happy and/or to avoid time lost waiting for slow IO

“*dealing with lots of things at once*”

- *parallelization*: to divide a task in subtasks that can be done in parallel, and assigning these subtasks to different processors (cores) to speed things up

“*doing lots of things at once*”

Is all about *optimization*, consider carefully whether really needed: complex and hard to test/debug.

Main consideration: is my problem CPU-bound or IO-bound?

117

## Concurrent execution: three approaches

1. *multiprocessing*: assign task to different processes; a process is instance of a running program (in this case: different instances of the interpreter, each running in their own space: no shared memory)
2. *threading*: a flow of control that shares memory (global state) with any other threads running; no control over scheduling of threads: *pre-emptive scheduling*
3. *event loop*: relies on co-routines, functions that can be halted, release control and continue where they left off when control is returned; *cooperative scheduling*

118

## Threads

use `threading` module

`threading.Thread(name=None, target=None, args=(), kwargs={})`

methods:

`start`: get thread started, i.e. calls `t.run()`, which by default calls `target`

`run`: can be overridden in subclasses of `Thread`

`join(timeout=None)`: `t.join()` blocks calling thread until `t` finishes (or at timeout)

`daemon`: set `t.daemon = True` to allow process to terminate even if `t` is still running

119

## Threads

Threads have their own stack, but share code and heap. Handy, but also dangerous: what if two threads make non-atomic changes to same object at same time?

Code is *thread-safe* if its results are guaranteed to be unaffected by any interruptions.

Ways to keep threads safe:

- only use atomic steps
- use only local data (stack), or immutable data (heap)
- use a *mutex* (mutual exclusion), a *lock* on critical sections of code

120

## Lock

- if a thread/process tries to acquire a lock, it blocks(waits) until lock comes available
- locks are unique objects, of two types: Lock or RLock
- RLock is reentrant: if already owned, no blocking (just increases counter)
- good protection, but tricky: what if someone forgets to release his lock?
- always use the lock's context manager, i.e. with `<lock>: statements`
- can protect against data corruption, but tricky:
  - deadlock: I need your key to continue, you need my key
  - starvation: how to ensure all processes have enough access to the locks

121

## Condition

- wraps a Lock (a new RLock when no existing one passed to constructor)
- adds a few methods (besides acquire and release)
- `c.wait(timeout=None)`: release lock (must own condition), wait for notification or timeout to retry acquire
- `c.notify()` or `c.notify_all()` to awake one or all waiting processes
- always use Condition's context manager, i.e. with `<condition>: statements`

```
with c:
    while not is_ok_state(s):
        c.wait()
    do_some_work_using_state(s)
```

```
with c:
    do_something_that_modifies_state(s)
    c.notify() # or, c.notify_all()
    # c is released when leaving with block
```

122

## Other synchronizers

**Event:** any number of threads can wait on an event e, event “happens” if e.set() is called

**Semaphore(n=1):** to manage limited set of resources; if no more available, acquire blocks or returns False, release wakes up a waiting thread or increases n again

**Barrier:** processes that wait on a barrier are resumed when specified number is reached

**Timer:** calls callable after specified interval, on new thread

```
class Periodic(threading.Timer):
    def __init__(self, interval, callable, args=(), kwargs={}):
        self.callable = callable
        threading.Timer.__init__(self, interval, self._f, args, kwargs)
    def _f(self, *args, **kwargs):
        Periodic(self.interval, self.callable, args, kwargs).start()
        self.callable(*args, **kwargs)
```

123

## Thread Local Storage

- a TLS object can store arbitrary attributes
- threadsafe
- each thread uses its own version
- no need to adapt ‘single-threaded’ code

```
import threading
loc = threading.local()
loc.foo = 42
def targ():
    loc.foo = 23
t = threading.Thread(target=targ)
t.start()
t.join()
```

124

## Queue

Queue: First-In, First-Out (FIFO) queue

LifoQueue: LIFO (Last-In, First-Out)

PriorityQueue: smallest first; generally uses pairs (priority, payload) as items

- `get(block=True, timeout=None)` return item or wait till one becomes available; if timeout is reached, or block is False, raises Empty exception; increases (hidden) *task counter*
- `put(block=True, timeout=None)` place item or if full, wait till room available; if timeout is reached, or no block is False, raises a Full exception
- `q.join()` blocks calling thread until no more tasks (call `q.task_done()` when done!)

125

## EAFP

Queue provides methods to get state:

`q.empty()` -> bool

`q.full()` -> bool

However, multi-threading means these cannot be guaranteed to be (or stay) correct. So do not try to Look Before You Leap but stick to EAFP (*Easier to Ask Forgiveness than Permission*)

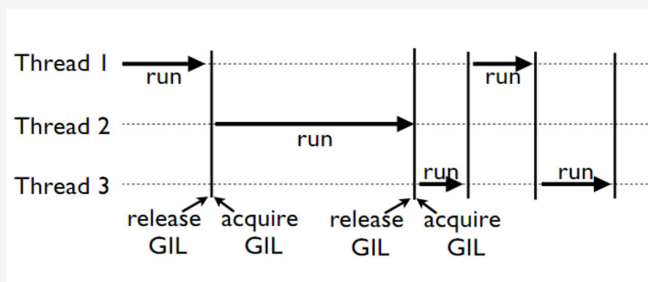
```
if q.empty():
    print('no work to perform')
else:
    x = q.get_nowait()
    work_on(x)
```

```
try:
    x = q.get_nowait()
except queue.Empty:
    print('no work to perform')
else:
    work_on(x)
```

126

## GIL – Global Interpreter Lock

Python itself is not thread-safe: allows non-atomic statements such as augmented assignment  
So interpreter itself uses a mutex or lock: only the thread that owns the lock can proceed



- *extensions (written in C) such as numpy can still be thread-safe*
- *GIL only protects single bytecode instructions*

127

## Threads vs. processes

- processes are managed by OS, run in their own space: no interference, no locking up
- BUT: requires IPC (inter process communication) to share data
  - e.g. pipes, files, databases, network connections, memory-mapped files
- much in common, fairly easy to switch between the two
- easiest when using `concurrent.futures`
- main differences:
  1. child processes *must* be able to import main script that's spawning them
  2. processes can only exchange objects that can be serialized ("pickled")

128



## multiprocessing

- use test harness to guard top-level code in main script
- `Process` is like `Thread` with few extra's:
  - `terminate` (but beware), `pid`, `exitcode`
- `multiprocessing.Queue` FIFO, no `join()/task_done()`, for this use `JoinableQueue`
- various Shared Object types:
  - `Value` (typecode (as for array type), \*args, lock=True)  
set/get value using value attribute
  - `Array` (typecode, size\_or\_initializer, lock=True)
  - `Manager`: a separate process providing proxies for exchanging objects

129

## multiprocessing

`Pool(processes=None, initializer=None, initargs=(), maxtasksperchild)`  
 reuses limited number of processes, default is `os.cpu_count()`

- `apply(func, args=(), kwds={})`
- `(i)map(func, iterable, chunksize=1)`
- `apply_async(func, args=(), kwds={}, callback=None)`
- `map_async(func, iterable, chunksize=1, callback=None)`
- async methods return immediately with an `AsyncResult`:
  - blocking: `get(timeout=None)`, `wait(timeout=None)`,
  - non-blocking: `ready()`, `successful()` *Note: not ready: AssertionError, False if error*

130

## concurrent.futures

- abstraction over multiprocessing and threading modules
- ThreadPoolExecutor/ProcessPoolExecutor: uses pool of threads/processes
- for each submitted task, executor returns a Future: a (pending) result
- a future holds *result* when set,
- can be given a *callback* (`add_done_callback`) for when done,
- has state of call (running/cancelled/done)
- `as_completed(fs, timeout)` returns iterator that generates futures as they complete
- easy to swap executors

131

## Run other programs

- `os` module: many methods starting with 'exec' to run executable file (unix systems)
- `popen(cmd, mode='r', bufferings=-1)` runs `cmd` in new process, returns file-like object to write data into, or results from, that process
- `system(cmd)`
- `subprocess` module

132

## Asynchronous processing

133

### Asynchronous processing

switch between tasks (when waiting for IO, or to divide attention fairly, etc.)

requires some form of scheduling:

- preemptive (as in threads): maybe too early, data corruption, deadlock, starvation
- event-based: task switched when something happens (mouse click, message arrived, computation ready ...)
- different ways to proceed after event:
  - make a call to a function (a call-back architecture), or:
  - continue a previously interrupted function (coroutine architecture)

134

## Coroutine

an object capable of:

- suspending,
- preserving state,
- explicitly handing back control to a scheduler (event loop),
- resuming when called upon

a coroutine *function*:

- returns a coroutine object
- executes immediately
- runs no user code

135


## From generator to coroutine

- generators can stop, saving state and restart where they left off
- for a coroutine we need one more option:
  - a way to get data *into* generator
- solution:
  - allow use of `yield` in expression
  - `send` method to inject value to yield expression

suspends at `yield` (after yielding `n` but before assigning `val`), so first need a `next` to get there:

```
> c = countdown(5)
> next(c)      -> 5
> c.send(3)    -> 2
```

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



136

## From generator to coroutine

Two more options were added:

- `close()` - to close coroutine
- `throw(ex, val, trace)` - to raise an exception inside generator/coroutine

```
#close method:
def close(self):
    try:
        self.throw(GeneratorExit)
    except (GeneratorExit, StopIteration): pass
    else:
        raise RuntimeError("generator ignored GeneratorExit")
    # Other exceptions are not caught
```

137

## Generators ≠ coroutines (just similar)

- historical relation: generator technology is used for coroutines, but:
- generators *produce* data for iteration
- coroutines are like functions: *consuming* inputs in addition to producing results
- coroutines can run in a single thread: cheaper and safer
- nice way to resolve blocking IO calls: process can do other stuff and continue later
- main application area for coroutines is IO, hence `asyncio`
- beats dealing with call-backs: these lead to spaghetti code (no state kept)

138

## Coroutine

two new keywords: `async` and `await`

`asyncio` package that provides the API's needed to:

- run coroutines concurrently
- perform network IO and IPC
- control subprocesses
- distribute tasks
- synchronize concurrent code

```
import asyncio
async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')
# Python 3.7+
asyncio.run(main())
```

139

## using asyncio

- coroutines are defined using `async def`
- a coroutine can only call another coroutine in an `await` statement
- an `await` expects an `Awaitable`: a coroutine, `Task` or `Future`
- without an `await` a coroutine will just run and return
- coroutines can only be called from other coroutines or by running them on event loop
- event loop is provided by `asyncio`
- `asyncio.create_task(coro())` schedules execution of task

140

## using asyncio

- works by using Future's, objects that can be immediately returned and act as a promise or placeholder for when the actual results arrive
- a coroutine should "yield", i.e. "await" when it suspects it might block, or just to give another coroutine a chance to run
- the coroutine sets its next result on the last future yielded when ready
- which triggers the Task driving the coro to take the next step as soon as event loop allows

141

## Future

A future result, waiting for a `set_result` call; this will in turn call any callbacks set  
 Note: `self.result` blocks

```
class Future:
    def __init__(self):
        self.result = None
        self._callbacks = []

    def add_done_callback(self, fn):
        self._callbacks.append(fn)

    def set_result(self, result):
        self.result = result
        for fn in self._callbacks:
            fn(self)
```

142

## Task

Drives coroutine by means of callbacks added as results get set

```
class Task:
    def __init__(self, coro):
        self.coro = coro
        f = Future()
        f.set_result(None)
        self.step(f)

    def step(self, future):
        try:
            next_future = self.coro.send(future.result)
        except StopIteration:
            return

        next_future.add_done_callback(self.step)
```

143

## Eventloop

task terminates when callback raises StopError when task.result is set

```
class EventLoop:
    def run_until_complete(self, coro):
        """Run until the coroutine is done."""
        task = Task(coro)
        task.add_done_callback(stop_callback)
        try:
            self.run_forever()
        except StopError:
            pass

class StopError(BaseException):
    """Raised to stop the event loop."""

def stop_callback(future):
    raise StopError
```

144



```

import time
import asyncio
async def main():
    print(f'{time.ctime()} Hello!')
    await asyncio.sleep(1.0)
    print(f'{time.ctime()} Goodbye!')
    loop.stop()
loop = asyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
pending = asyncio.Task.all_tasks(loop=loop)
group = asyncio.gather(*pending, return_exceptions=True)
loop.run_until_complete(group)
loop.close()
Output:
$ python quickstart.py
Sun Sep 17 14:17:37 2017 Hello!
Sun Sep 17 14:17:38 2017 Goodbye!

```

145

## Context variables

- modules such as logging or decimal use global data structures (configurations or context); while these are generally threadsafe, this is not sufficient for asynchronous processing
- ContextVars can be used to restrict settings to specific coroutines
- asyncio has been adapted to use these contexts: Task creates them, various loop calls take them as argument

decimal.getcontext().prec = 4  
with decimal.localcontext() as c:  
    c.prec = 2

```

var = ContextVar('var')
var.set('spam')
def main():
    var.set('ham')
ctx = copy_context()
ctx.run(main)
ctx[var] == 'ham'
var.get() == 'spam'

```

146

## Serialization and persistence

147

### Serialization and persistence

- to save Python objects to byte streams, and restore them again from those streams
- options: JSON (general), pickle (Python specific), XML, YAML, [protocol buffers](#) ....)
- `shelve` combines pickle with tools for storing key-value pairs
- a `Shelf` is just like a dictionary, with all entries kept in sync with its key-value store on file
- other option: a database, using a third-party module conforming to Python Database API 2.0

148

## JSON = Javascript Object Notation

```
import json
json.dumps(['a', None, {'k': 23}]) → "[ 'a', null, { 'k': 23 } ]"
json.loads("[ 'a', null, { 'k': 23 } ]") → [ 'a', None, { 'k': 23 } ]
```

- universal and compact
- choose separators
- pretty printing
- add your own decoding hooks (functions)
- subclass JSONEncoder and/or JSONDecoder

149

## pickle

```
>>> import pickle
>>> with open("pickled", 'wb') as f:
>>>     pickle.dump([1,2], f)
>>> with open("pickled", 'rb') as f:
>>>     l = pickle.load(f)
```

[dumps / loads to skip file IO]

Differs from e.g. JSON:

- shared objects remain shared (handles recursive data)
- save and restore instances (requires import of class definition!)
- compatible across Python versions
- specific for Python: binary, not human-readable

150

## Python DB API

- Most Python database interfaces adhere to this standard
- Each database has its own module highly consistent with others
- Modules available for most of the popular relational databases
- Python DB API 2.0 compatible modules:
  - `sqlite3` part of the standard library: the SQLite database is part of installation
  - `mysql` (MySQL) – MySQL Connector/Python
  - `psycopg2` (PostgreSQL)
  - `cx_Oracle` (Oracle)
  - `pyodbc` or `mxdmdbc` for ODBC-compliant databases (most are)

**SQLite:** a self-contained (single file), server-less, zero-configuration, transactional SQL database engine

151

## DB API Concepts

Two main concepts (classes): `Connection` and `Cursor`

`Connection` objects logically wrap a database connection:

- network/RPC access to the database
- a means to handle database transactions
- ex: `connection = sqlite.connect('sample.db')`

`Cursor` objects represent results:

- created by `cursor()` method of `connection` instance
- used to execute statements and fetch records
- `cursor = connection.cursor()`
- `cursor.execute('create table testtable(id int, name varchar(254))')`

152

## DB API cont.

```
cursor.execute(statement)
```

```
cursor.executemany("INSERT INTO table VALUES(?, ?, ?)", tuples)
```

```
cursor.fetchone / cursor.fetchmany / cursor.fetchall
```

sqlite3.Row can be set as connection.row\_factory: row as dictionary

- to write all changes of the last transaction block to the database:

```
conn.commit()
```

- to undo all changes applied to the database on the connection:

```
conn.rollback()
```

with statement to handle commit (and rollback in case of error)

153

## XML

XML (eXtensible Markup Language): used to exchange data (mostly text) that is tagged, given attributes and organized in a hierarchical, tree-like structure

```
<courses>
  <course category="XML">
    <code>XML801</code>
    <title>XQuery</title>
    <price>2200</price>
    <duration>2</duration>
    <year>2008</year>
    <location>Amsterdam</location>
  </course>
</courses>
```

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Title of the document</title>
</head>
<body>
  Content of the document.....
</body>
</html>
```

154

## XML Processing Options

The DOM (Document Object Model):

API for HTML and XML documents. It represents the document as nodes and objects that be used to inspect and change the document structure, style, and content. W3C standard, used by most browsers. Disadvantage: complete document must be kept in memory.

XML handling submodules:

- [xml.etree.ElementTree](#): the ElementTree API, a lightweight XML processor
- [xml.dom](#): the DOM API definition
- [xml.dom.minidom](#): a minimal DOM implementation
- [xml.dom.pulldom](#): support for building partial DOM trees
- [xml.sax](#): SAX2 base classes and convenience functions
- [xml.parsers.expat](#): the Expat parser binding

155

## XML Processing options

The main distinction:

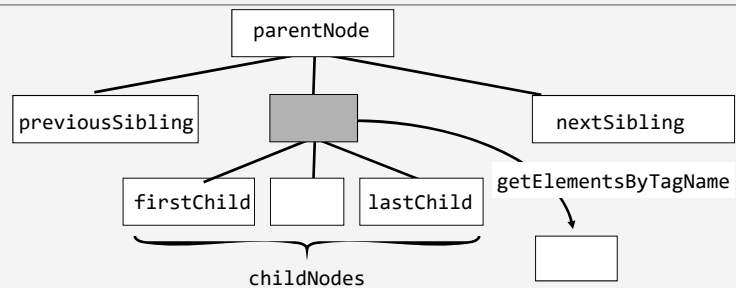
1. keep complete document in memory, search over tree structure (DOM)
2. treat document as a stream, reacting to events such as *start element*, *end element*

The streaming approach can take two forms:

1. callbacks, functions (methods) called by a ContentHandler given to a parser:  
parser just produces elements; user needs to keep track of state (SAX approach)
2. **pull** elements from stream: similar in that parser produces stream of events, but  
when e.g. an interesting tag is encountered, you can pull in the complete element:  
no need to keep track of state

156

## DOM Navigation



Property	Description
childNodes	A collection of node objects that are immediate children of the owning node
firstChild	The first item in the childNodes collection
lastChild	The last item in the childNodes collection
parentNode	A reference to the node whose childNodes collection this node belongs to
previousSibling	Node immediately before current node in childNodes collection belonging to their joint parent node
nextSibling	Node immediately after current node in childNodes collection belonging to their joint parent node
getElementsByTagName (tagName)	Returns a collection of objects having been instantiated from the specified tag name

157

## Sax Callbacks

```

import xml.sax
class HandleCollection (xml.sax.ContentHandler):
    # Called at the start of an element
    def startElement (self, name, attributes):
        print("Inside startlement: " + name)
        print(*attributes.items())
    # Called at the end of an element
    def endElement (self, name):
        print("Inside endlement: " + name)
    # Called to handle content besides elements
    def characters (self, content):
        print("Inside characters: " + content)
parser = xml.sax.make_parser()
parser.setContentHandler (HandleCollection())
parser.parse ('courses.xml')
  
```

158

## Reusing code

159

### Modules

- code and data organized in source files called modules
- modules: reused in other modules using `import` or `from` statements
- no really global variables, just attributes of a module object
- modules can be scripts, or *extension* modules, coded in C or other language
- same import statement, so easy to switch script for compiled C version
- modules can be hierarchically organized in packages, tree-like structures
- once imported, modules are *first-class objects*, inspectable, mutable etc.

160



## Import

- Python scripts (or extensions) can be imported using import statement:
  - `import module1[, module2[, ... moduleN]`
- import succeeds if *module.py* found on `SYS.PATH`, a list of directories
- imported script is executed, bindings added to module object
- imports are skipped if module was already imported  
to reload: `importlib.reload(module)`
- alias: `import test as t`

161

## import Statement

- to add names directly to importing namespace:
  - `from a_mod import foo [as baz][, ...]`      Rebinds existing name(s) without warning!
- `from a_mod import *` will import all names
  - generally bad idea (just use short alias if typing fatigue is the issue)
- import creates module object (with e.g. `__name__`, `__doc__`, `__package__`, `__dict__`)
- `__dict__` holds bindings of all names defined in module
- `dir(mod) == mod.__dict__.keys().sort()`
- `__doc__` is filled if first statement is a literal string
- convention: names starting with underscore are intended to be private

162

## built-ins

- built-ins are objects provided by Python, such as list, dict
- these are attributes of the builtins module
- each module gets a `__builtins__` dict that holds the builtins
- name lookup: first local and global namespace, then `__builtins__`
- easy to rebind names of built-ins
- can add or substitute built-ins for all modules by importing builtins module and rebinding name in that module

```
import builtins
_old = builtins.term
def _new(...):
    statement(s) # wrapping _old?
builtins.term = _new
```

163

## files loaded by import

if module already loaded (exists in `sys.modules`): return that module, else look for:

1. extension modules: files ending on `.dll` or `.pyd` (Windows), `.so` (Unix)
  2. source: `.py` file; if found, look in `__pycache__` folder for proper `.pyc` version
  3. bytecode compiled: `.pyc`
- only imported module files are compiled to `.pyc`
  - `importlib.reload(module_object)` reassigns names, does not replace objects; it also is not recursive, so imports in reloaded are not reloaded also
  - avoid circular imports

164

## test harness

modules meant for import (reuse) should ideally contain only binding statements:

assignments and definitions (of functions, classes etc.)

use test harness to restrict execution to when module is the top (main) script:

```
if __name__ == '__main__':  
    statements
```

165

## Packages

- a collection of modules and/or subpackages
- resides in a subdirectory of a directory (or a zip file) on sys.path
- `from package import module`
- `import package.module`
- `from package.subpackage.module import name`
- within package: `from .subpackage import module`
- within subpackage: `from ..other_subpackage import module`

166

## Packages

- package P *may* have module body: always called `__init__.py` in P
- P module (i.e. `__init__.py`) is imported whenever P or a module in P is imported
- `__init__.py` can be used to make names from its modules appear as package names:  
`from module import x, y, z →`
- if no `__init__.py` found in folder, it is called a *namespace package*

167

## Distribution

distribution of libraries:

1. *setuptools* and *wheels* to create distribution (archive plus setup script)
2. *twine* to upload it to some repository, e.g. PyPI
3. *pip* to install

alternatives:

- containers, such as Docker
- app platforms such as Heroku
- *pipx* (to install python apps: in virtual environment, callable from CLI)
- *conda*
- see Python Packaging User Guide (<https://packaging.python.org>)

168

## Creating Packages

1. `pip install -U pip setuptools wheel twine` [sample guide to PyPi](#)
2. create `setup.py` at root of project folder: [sample setup file](#)

```
from setuptools import setup, find_packages
setup(named arguments giving details of project: name, description, author, license,
homepage, classifiers, packages included or required, entry-points etc. )
```
3. make distribution (a `.tar.gz` file)

```
python setup.py sdist (source only, so may require C compiler)
python setup.py bdist_wheel --universal (python only)
python setup.py bdist_wheel (compiled extensions)
```
4. `twine register` # may need to register package before uploading  
`twine upload -r repo dist/*` # repo details in `~/.pypirc` config

169

## Virtual environments

- Each environment has its own Python binary and packages
- Always one environment active
- Independent: changes to environments do not affect each other

```
python -m venv myenv c:\path\to\myenv
c:\path\to\myenv\Scripts\activate
(myenv) :> deactivate (to delete just remove c:\path\to\myenv)
```

### Easy to create distributions

- to distribute: `pip freeze >> requirements.txt`
- to install: `pip install -r requirements.txt`

170

## Standard library

171

### Standard library: built-in types

- operator module, function equivalent for Python's operators
- random
- `os.urandom`
- `heapq`  
invariant when adding/removing items:  
 $\text{heap}[k] \leq \text{heap}[2*k+1]$  and  
 $\text{heap}[k] \leq \text{heap}[2*k+2]$   
useful for e.g. priority queue

`functools`, e.g. `lru_cache`

0							
1				2			
3		4		5		6	
7	8	9	10	11	12	13	14

172

## Standard library: collections

- `collections.abc` – Abstract Base Classes for containers
- `ChainMap(*mappings)`: chain multiple mappings, for search , iteration etc.
- `Counter(iterable)`: a frequency table
  - all dict methods plus elements, `subtract(iterable)`, `most_common([n])`
- `OrderedDict` (keys in insertion order, **redundant in 3.7!**)
- `defaultdict`: provide some factory to provide default values

```
d = collections.defaultdict(list)
for key, value in items:
    d[key].append(value)
```

173

## Standard library: collections

- `deque(seq=(), maxlen=None)`: double-ended queue  
insert and append equally fast, when maxlen reached item dropped at other end
- `namedtuple(typename, fields)`  
methods: `_asdict`, `_fields`,  
`_make(seq)`, `_replace(**kwargs)`

```
point = collections.namedtuple('point', 'x,y,z')
p = point(x=1,y=2,z=3)
x, y, z = p # can unpack like a normal tuple
if p.x < p.y: print(x) # point(x=1, y=2, z=3)
```

174

## Standard library: Data Classes

- mutable namedtuples with defaults
- adds special methods to a class:
  - `__init__`, `__repr__`, `__eq__`, `__ge__`, etc.
- type hints on fields are required (use `Any` if anything goes ...)
- `field()` specifier to customize field
- `dataclass` decorator can be given parameters to control methods generated, mutability etc.

```
from dataclasses import dataclass
@dataclass
class InventoryItem:
    name: str
    unit_price: float
    stock: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.stock
```

175

## Standard library: typing

see PEP 484/526

annotations stored under `__annotations__`

- function and parameter type annotations
- not enforced by interpreter: use e.g. `MyPy`
- no effect whatsoever on run-time
- variable annotations (incl. class/instance attrs)

```
primes: List[int] = []

captain: str # No initial value!

class Starship:
    stats: Dict[str, int] = {}
```

```
def greeting(name: str) -> str: body
```

```
from typing import List
Vector = List[float]
```

```
from typing import NewType
UserId = NewType('UserId', int)
some_id = UserId(524313)
```

```
from typing import Sequence, TypeVar
T = TypeVar('T')
def first(l: Sequence[T]) -> T:
    return l[0]
```

176



## Standard library: argparse

to handle CLI arguments to a script

```
import argparse
ap = argparse.ArgumentParser(description='An example')
ap.add_argument('who', nargs='?', default='World')
ap.add_argument('--formal', action='store_true')
ns = ap.parse_args()
if ns.formal:
    greet = 'Most felicitous salutations, o {}.'
else:
    greet = 'Hello, {}!'
print(greet.format(ns.who))
```

177

## Garbage collection and weak references

- when ref count == 0, Python call `obj.__del__()` and frees memory
- issue: cyclic references (e.g. items with reference to their container)
- gc module looks for objects that only refer to each other
  - can be disabled (and enabled again)
  - allows inspection (e.g. if to `__del__` methods prevent collection)
- solution for e.g. caches is *weak* reference (from `weakref` module)
- `weakref.proxy(x[, f])` a proxy `p` for `x`
- refcount of `x` is not incremented when held by proxy

178

# Logging

179

## Logging messages

To pinpoint problems it helps to log specific events:

1. simply add (and then remove) print statements
2. use logging

Advantages:

- distinguish different levels for different purposes (auditing, debugging etc.)
- easy to turn on/off
- print to console, file, send mail etc.
- change format (what to include and how)

180

## Events (messages) differ in importance / severity

<i>Level</i>	<i>Numeric Value</i>	<i>Function</i>	<i>Used to</i>
CRITICAL	50	logging.critical()	Show a serious error, the program may be unable to continue running
ERROR	40	logging.error()	Show a serious problem
WARNING	30	logging.warning()	Indicate something unexpected happened, or could happen
INFO	20	logging.info()	Confirm that things are working as expected
DEBUG	10	logging.debug()	Diagnose problems, show detailed information

- default level is WARNING

181

## logging

```
import logging
class Foo():
    def __init__(self, a):
        self.a = a
        logging.debug('Foo created: {}'.format(self.a))
```

- with default configuration (WARNING) no debug log written !
- to change level in configuration:
- change configuration before any logging  
(note: basicConfig can only be called once!)
- to write to file instead of console: logging.basicConfig(filename='log.txt')

182

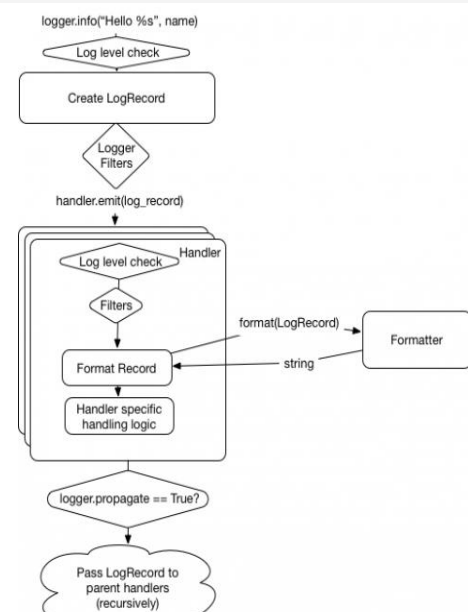
## Components of logging

- **loggers:** instances of Logger class, provides log methods
  - have names, arranged in hierarchical namespace
  - can give each module its own logger to reflect package/module hierarchy  
`logger = logging.getLogger(__name__)`
- **handlers:** handle different destinations (files, HTTP, email, sockets, queues, syslog etc.); make subclass for new destinations ); multiple handlers on logger for e.g. different levels
- **filters:** determine whether logger or handler passes on logrecord, based on other info than level; can also modify record
- **formatters:** determine format of record

183

## Process flow

- logging call
- logger enabled for level?
- create logRecord
- if filter attached, does record pass?
- pass to handlers of current logger
  - handler enabled for level?
  - filter OK?
  - format and emit
- OK to propagate and has parent?
- pass to handlers of parent



184

## Logger.addHandler(handler)

StreamHandler	streams (file-like objects).
FileHandler	disk files.
BaseRotatingHandler	handlers that rotate log files at a certain point
RotatingFileHandler	disk, with support for maximum file sizes and rotation.
TimedRotatingFileHandler	disk files, rotating the log file at certain timed intervals.
SocketHandler	TCP/IP sockets. Since 3.4, supports Unix domain sockets
DatagramHandler	UDP sockets. Since 3.4, supports Unix domain sockets
SMTPHandler	designated email address.
SysLogHandler	a Unix syslog daemon, possibly on a remote machine.
NTEventLogHandler	a Windows NT/2000/XP event log.
MemoryHandler	a buffer in memory, flushed whenever criteria are met
HTTPHandler	HTTP server using either GET or POST semantics.
WatchedFileHandler	instances watch the file they are logging to
QueueHandler	using a queue, as in <i>queue</i> or <i>multiprocessing</i> modules

185

## Formatting

LogRecords store name of logger, level, pathname and line of source, calling function etc.

- create own LogRecordFactory to include other info
- default format: <LEVEL>:<loggername>:<message>
- set format string using appropriate attributes from record:

```
logging.basicConfig(format = '%(asctime)s %(message)s')
```

```
or: root = logging.getLogger()
```

```
    root.setLevel(logging.DEBUG)
```

```
    handler = logging.StreamHandler()
```

```
    bf = logging.Formatter('{asctime} {name} {levelname:8s} {message}')
```

```
    handler.setFormatter(bf)
```

```
    root.addHandler(handler)
```

186

## Unit testing

187

### Unit testing

- determine the correctness of a single module, class or function
- pre-condition for *integration* or *systems* testing
- basis of test-driven development:
  1. write a test (a runnable specification)
  2. implement while regularly testing
  3. if code that passed test is later found to fail:
    - a. first debug test to make test also fail
    - b. only then debug code itself
- crucial to keep tests fast and reliable:  
*stubs / mocks / dummies / fakes* for interfaces

frameworks for automating tests:

- [unittest](#): built-in standard library
- [pytest](#): complete framework
- [nose](#): an extension to unittest
- [hypothesis](#): a unit test-generation tool
- [doctest](#): run examples in doc

Modules / Packages

188

## doctest

- idea: good documentation gives examples of proper and improper usage
- find examples and run them
- advantages
  1. easy way to run tests
  2. guarantees doc stays up-to-date
  3. tests become part of explanation
  4. can even produce unittest suites
- need to balance good documentation with enough examples to cover all cases

```
def factorial(n):
    """Return the factorial of n, an
    exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    26525285981219105863630848000000
    >>> factorial(-1)
    Traceback (most recent call last):
    ...
    ValueError: n must be >= 0
    """

    statements

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

189

## unittest

- a test case is a subclass of TestCase
- a testcase is instantiated and all test\_ methods are run
- these methods can use TestCase methods starting with assert
- can override setUp and tearDown methods that are run before/after each test (TestCase also maintains LIFO stack of clean-up functions)

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # s.split should fail if separator not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

Modules / Packages

190

## Prepare and tidy up

- override setUp and tearDown methods to run stuff before / after each test
- TestCase also maintains LIFO stack of clean-up functions:
- tc.addCleanup(func, \*a, \*ka)
- tc.doCleanups()- happens automatically after tearDown

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def testSomething(self):
        statements

    def tearDown(self):
        self.widget.dispose()
```

191

## unittest: test discovery

- keep TestCases in separate modules
  - name them as “test\_<name of module tested>”
  - allows for automatic discovery:
- ```
cd project_directory
python -m unittest [discover]
```
- Note: all of the test files must be modules or packages (including namespace packages) importable from the top-level directory of the project
- discover options:
- s, --start-directory *directory* (default .)
  - p, --pattern *pattern* (default test\*.py)
  - t, --top-level-directory *directory* (defaults to start directory)

Modules / Packages

192



## Monkey Patching

Python code which extends or modifies other code at runtime (typically at startup):

```
from SomeOtherProduct.SomeModule import SomeClass
def speak(self):
    return "eee eee eee!"
SomeClass.speak = speak
```

Decorators are another example of monkey patching:

```
import datasource
def get_data(self):
    '''monkey patch datasource.Structure to simulate error'''
    raise datasource.DataRetrievalError
datasource.Structure.get_data = get_data
```

193

## Cython

194

## Cython

1. language: Python with static typing
2. compiler: compiles Cython code to C or C++, which can in turn be compiled into exe or Python extension;
3. also used to turn existing C libraries into Python extensions
  - faster function call (less overhead)
  - faster math operations (type specific)
  - faster looping (optimized by compiler)
  - faster due to less use of heap

Cython, Kurt W. Smith, 2015, O'Reilly

195

## Extension from C code:

1. cfib.h: `double cfib(int n);`
2. Cython wrapper function:
 

```
cdef extern from "cfib.h":
    double cfib(int n)
def fib(n):
    "Returns nth Fib number."
    return cfib(n)
```
3. compile to wrap\_fib extension
4. `>>> from wrap_fib import fib`

196

## Extension from Cython code

```
1. in fib.pyx:
def fib(int n):
    "Returns nth Fibonacci number."
    cdef int i
    cdef double a=0.0, b=1.0
    for i in range(n):
        a, b = a + b, a
    return a

2. in setup.py:
from distutils.core import setup
from Cython.Build import cythonize
setup(ext_modules=cythonize('fib.pyx'))

3. python setup.py build_ext -i --compiler=msvc

4. import fib
```

197

## Debugging

198

## Debugging

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as you can, you are, by definition, not smart enough to debug it." *Brian Kernighan*

- first update test cases
- inspect module: functions to get information about live objects, including about frames
- traceback: `print_exc(limit=None, file=sys.stderr)` in exception handler prints the traceback generated for uncaught exceptions
- pdb a simple command-line interactive debugger:
  - `pdb.run(some_code)`
  - `pdb.pm()` – post-mortem after crash
  - `pdb.set_trace()` in code to trigger pdb at that point (breakpoint)

199

## pdb

- `n(ext)` - execute current line (do not step "into" if function, except for breakpoints)
- `s(tep)` - next, but step into functions
- `p <exp>` - print expression
- `! statement` - execute statement
- `c(ontinue)` - execute until next breakpoint or end
- `b(reak) [location [, condition]]`
- `cl(ear)`
- `u(p)` - go back to previous frame
- `d(own)` - back down to later frame
- `ignore breakpoint-nummer count` - skip this breakpoint *count* times

200

## optimization, profiling and timing

focus on algorithms and on inherent properties of data types, e.g.

- lists: append is  $O(1)$ , insert is  $O(N^2)$
- search in sets and dicts is  $O(1)$ , in lists  $O(N)$
- string: repeated concatenation ( $s+s+s+s+..$ ) is  $O(N^2)$
- (waiting for) IO can have huge impact (solution: see asynchrony)

minor stuff:

- access to local variables is faster than to globals
- attribute access takes time etc.
- `timeit` shows overall time spent in statement(s)
- profiling, e.g. with `cProfile` or `IPython` magic, shows where most time is spent

201

## Boilerplate properties

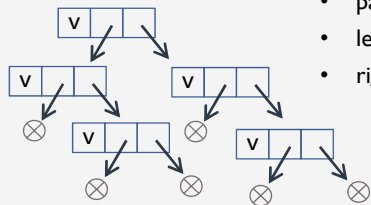
```
def typed_prop(name, p_type):
    storage_name = '_' + name
    @property
    def prop(self):
        return getattr(self, storage_name)
    @prop.setter
    def prop(self, value):
        if not isinstance(value, p_type):
            raise TypeError('{} must be a {}'.format(name, p_type))
        setattr(self, storage_name, value)
    return prop
```

```
class Person:
    name = typed_prop('name', str)
    age = typed_prop('age', int)
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
from functools import partial
String = partial(typed_prop, p_type=str)
Integer = partial(typed_prop, p_type=int)
class Person:
    name = String('name')
    age = Integer('age')
```

202

## Recursion



Tree of 3-tuples:

- payload (v)
- left subtree
- right subtree

- Python is not clever about recursion
- default recursion limit: 1000
- try to find no-recursive solutions

```
def rec(t):
    yield t[0]
    for i in (1, 2):
        if t[i] is not None:
            yield from rec(t[i])

def norec(t):
    stack = [t]
    while stack:
        t = stack.pop()
        yield t[0]
        for i in (2, 1):
            if t[i] is not None:
                stack.append(t[i])
```

203

## Patterns

204

## What about patterns?

Pattern: a general solution for a type of problem

In software engineering:

*Design Patterns: Elements of Reusable Object-Oriented Software (1994)*

Creational patterns:

Factory, Singleton, Dependency Injection ...

Structural patterns:

Adapter, Decorator, Proxy, ...

Behavioral patterns:

Iterator, Observer, Command, Chain of responsibility ...

205

205

## Observer pattern

Variants: *publish and subscribe* or *produce and consume*

Decouple processes: producers and consumers of information do not need to know of each other. A publisher just publishes, a subscriber just subscribes. Subscribing means being put on a list and being informed when something is published, publishing means informing the subscriber of a publishing event, perhaps through some broker.

*Examples*:

- GUI components register interest in mouse clicks
- news readers register interest in news (on specific topics)
- Broker will generally call some registered function (callback)

206

## Patterns or principles?

- program to an interface, not an implementation
- favor composition over inheritance

Façade (hide parts of an API): create class that does just that

Adapter: make class conform to some interface by wrapping

Decorator: covered

Iterator: covered

Dependency Injection: comes for free

*dependent:*

```
class A:
    def __init__(self, component):
        self.component = Component()
```

*dependency "injected":*

```
class A:
    def __init__(self, component):
        self.component = component
```

207

JdB1

## Chain of responsibility

Every piece of code must do one, and only one, thing.

```
class ContentFilter(object):
    def __init__(self, filters=None):
        self._filters = list()
        if filters is not None:
            self._filters += filters
    def filter(self, content):
        for filter in self._filters:
            content = filter(content)
        return content

filter = ContentFilter([
    offensive_filter,
    ads_filter,
    porno_video_filter])
filtered_content = filter.filter(content)
```

208

208





## Command: request as object

```
class RenameFile(object):
    def __init__(self, old, to):
        self._from = old
        self._to = to
    def execute(self):
        os.rename(self._from, self._to)
    def undo(self):
        os.rename(self._to, self._from)
```

```
class History(object):
    def __init__(self):
        self._commands = list()
    def execute(self, command):
        self._commands.append(command)
        command.execute()
    def undo(self):
        self._commands.pop().undo()

history = History()
history.execute(RenameFile('cv.doc', 'cv-en.doc'))
history.execute(RenameFile('cv1.doc', 'cv-bg.doc'))
history.undo()
history.undo()
```

209

209

## EAFP – Easier to ask forgiveness than permission

accept that exceptions will occur and handle those when they do

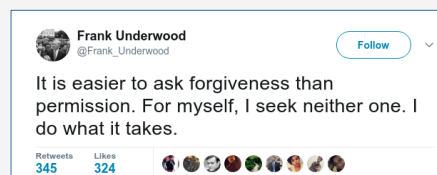
so instead of:

```
if hasattr(obj, 'attr'):
    obj.attr()
else:
    [...]
```

just try it:

```
try:
    obj.attr()
except AttributeError:
    [...]
```

- more efficient (if exceptions are the exception)
- cleaner and clearer code



210

## Duck typing

*"If it walks like a duck and quacks like a duck, it is a duck."*

Dynamic typing:

- an object's current set of methods and properties determines the valid semantics, not its type (i.e. its inheritance from a specific class or its implementation of a specific interface).
- types can be modified at run time, so compiler cannot check beforehand whether a method can actually be called on object
- getting this right is up to user; although developer could include run time type checking, e.g. by using `isinstance(obj, type)`

211

## DRY: Don't repeat yourself

- programmers (should) get bored easily: repetition suggests automation
- repetition is tedious, hard to read, hard to modify
- metaprogramming can help
- metaprogramming: code that manipulates code
- Python offers number of options:
  - decorators
  - metaclasses
  - descriptors

212

## Hashing

hash(x), e.g. if x is to act as key in a dict, calls x.\_\_hash\_\_()

if \_\_hash\_\_ is absent:

if \_\_eq\_\_ is absent:

calling hash(x) calls id(x)

else Exception is raised

hash(x) returns int, such that

- $x == y$  implies  $\text{hash}(x) == \text{hash}(y)$
- $a = \text{hash}(x)$  and  $b = \text{hash}(x)$  implies  $a == b$

213

```
class listNoAppend(list):
    def __getattr__(self, name):
        if name == 'append': raise AttributeError(name)
        return list.__getattr__(self, name)
```

214

# Networking

215

## IP Networks

- Datagram-based
  - Connectionless
- Unreliable
  - Best efforts delivery
  - No delivery guarantees

|                                                                                                |     |      |     |     |     |      |              |
|------------------------------------------------------------------------------------------------|-----|------|-----|-----|-----|------|--------------|
| Telnet                                                                                         | SSH | SMTP | FTP | NFS | DNS | SNMP | Application  |
| TCP                                                                                            |     |      |     | UDP |     |      | Host-to-host |
| IP                                                                                             |     |      |     |     |     |      | Internetwork |
| Ethernet,Token Ring, RS232, IEEE 802.3, HDLC, Frame Relay, Satellite,Wireless Links,Wet String |     |      |     |     |     |      | Subnetwork   |

216

## UDP Characteristics

Also datagram-based

- Connectionless, unreliable, can broadcast

Applications usually message-based

- No transport-layer retries

- Applications handle (or ignore) errors

Processes identified by port number

Services live at specific ports

- Usually below 1024, requiring privilege

217

## TCP Characteristics

Connection-oriented

- Two endpoints of a virtual circuit

Reliable

- Application needs no error checking

Stream-based

- No predefined blocksize

Processes identified by port numbers

Services live at specific ports

218

## Client/Server Concepts

Server opens a specific port

- The one associated with its service

- Then just waits for requests

- Server is the passive opener

Clients get ephemeral ports

- Guaranteed unique, 1024 or greater

- Uses them to communicate with server

- Client is the active opener

219

## Simple Connectionless Server

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
s.bind(('127.0.0.1', 11111))
data = b"data"
while data:
    data, addr = s.recvfrom(1024)
    print ("Received %r from %s " % (data, addr))
    s.sendto(data.upper(), addr)
```

220

## Simple Connectionless Client

```
from socket import socket, AF_INET, SOCK_DGRAM
srvaddr = ('127.0.0.1', 11111)      # server address
s = socket(AF_INET, SOCK_DGRAM)    # create a socket
s.bind(('127.0.0.1', 0))           # can also use ('', 0)
print("Socket:", s.getsockname())
data = b"string"
while data:
    data = bytes(input("Send: "), 'UTF-8') # gets data from user
    s.sendto(data, srvaddr)                # send the data
    respons, addr = s.recvfrom(1024)       # receive the reply
    print("Recv:", respons)
```

221

## Connection-oriented

### Server

```
from socket import socket, AF_INET, SOCK_STREAM
s = socket(AF_INET, SOCK_STREAM)
s.bind(('127.0.0.1', 9999))
s.listen(5) # max queued connections
while 1:
    sock, addr = s.accept()
    # use socket sock to communicate
    # with client process
```

### Client

```
s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
```

Client connection creates new socket  
Returned with address by accept()  
Server handles one client at a time

222