

# F21SC Industrial Programming: Python: Advanced Language Features

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,  
Heriot-Watt University, Edinburgh



Semester 1 2017/18

---

<sup>0</sup>No proprietary software has been used in producing these slides

# Overloading

- Operators such as `+`, `<=` and functions such as `abs`, `str` and `repr` can be defined for your own types and classes.

## Example

```
class Vector(object):  
    # constructor  
    def __init__(self, coord):  
        self.coord = coord  
    # turns the object into string  
    def __str__(self):  
        return str(self.coord)  
  
v1 = Vector([1,2,3])  
# performs conversion to string as above  
print (v1)
```

# Overloading

## Example

```
class Vector(object):
    # constructor
    def __init__(self, coord):
        self.coord = coord
    # turns the object into string: use <> as brackets, and ;
    def __str__(self):
        s = "<"
        if len(self.coord)==0:
            return s+">"
        else:
            s = s+str(self.coord[0])
            for x in self.coord[1:]:
                s = s+";"+str(x);
            return s+">"

v1 = Vector([1,2,3]); print (v1)
```

# Overloading arithmetic operations

## Example

```
import math      # sqrt
import operator  # operators as functions

class Vector(object):
    ...
    def __abs__(self):
        '''Vector length (Euclidean norm).'''
        return math.sqrt(sum(x*x for x in self.coord))
    def __add__(self, other):
        '''Vector addition.'''
        return map(operator.add, self.coord, other.coord)

print(abs(v1))
print(v1 + v1)
```

# Overloading of non-symmetric operations

- Scalar multiplication for vectors can be written either  $v1 * 5$  or  $5 * v1$ .

## Example

```
class Vector(object):  
    ...  
    def __mul__(self, scalar):  
        'Multiplication with a scalar from the right.'  
        return map(lambda x: x*scalar, self.coord)  
  
    def __rmul__(self, scalar):  
        'Multiplication with a scalar from the left.'  
        return map(lambda x: scalar*x, self.coord)
```

- $v1 * 5$  **calls** `v1.__mul__(5)`.
- $5 * v1$  **calls** `v1.__rmul__(5)`.

# Overloading of indexing

- Indexing and segment-notation can be overloaded as well:

## Example

```
class Vector(object):

    def __getitem__(self, index):
        '''Return the coordinate with number index.'''
        return self.coord[index]

    def __getslice__(self, left, right):
        '''Return a subvector.'''
        return Vector(self.coord[left:right])

print v1[2]
print v1[0:2]
```

## Exercise (optional)

- Define a class `Matrix` and overload the operations `+` and `*` to perform addition and multiplication on matrices.
- Define further operations on matrices, such as `m.transpose()`, `str(m)`, `repr(m)`.

# Types

- `type(v)` yields the type of `v`.
- Type-membership can be tested like this  
`isinstance(val, typ)`. E.g.  

```
>>> isinstance(5, float)
False
>>> isinstance(5., float)
True
```
- This check observes type-membership in the parent class. E.g.  

```
>>> isinstance(NameError(), Exception)
True
```
- `issubclass` checks the class-hierarchy.  

```
>>> issubclass(NameError, Exception)
True
>>> issubclass(int, object)
True
```

# Manual Class Generation

- `type(name, superclasses, attributes)` creates a class object with name `name`, parent classes `superclasses`, and attributes `attributes`.
- `C = type('C', (), {})` corresponds to `class C: pass`.
- Methods can be passed as attributes:

## Example

```
def f (self, coord):  
    self.coord = coord  
  
Vec = type('Vec', (object,), {'__init__' : f})
```

- Manual class generation is useful for **meta-programming**, i.e. programs that generate other programs.

# Properties

- *Properties* are attributes for which read, write and delete operations are defined.
- Construction:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

## Example

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    # this generates a read only property
    area = property(
        lambda self: self.width * self.height, # anonymous
        doc="Rectangle area (read only).")

print("Area of a 5x2 rectange: ", Rectangle(5,2).area)
```

# Controlling Attribute Access

- Access to an attribute can be completely re-defined.
- This can be achieved as follows:

```
__getattr__(self, attr)
__setattr__(self, attr, value)
__delattr__(self, attr)
```

- Example: Lists without append

## Example

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

# Static Methods

- A class can define methods, that don't use the current instance (`self`).
  - ▶ Class methods can access class attributes, as usual.
  - ▶ Static methods can't do that!.

## Example

```
class Static:
    # static method
    def __bla(): print ("Hello, world!")
    hello = staticmethod(__bla)
```

- The static method `hello` can be called like this:

```
Static.hello()
Static().hello()
```

# Class/Instance Methods

- A class or instance method takes as first argument a reference to an instance of this class.

## Example

```
class Static:
    val = 5
    # class method
    def sqr(c): return c.val * c.val
    sqr = classmethod(sqr)
```

```
Static.sqr()
Static().sqr()
```

- It is common practice to overwrite the original definition of the method, in this case `sqr`.
- **Question:** What happens if we omit the line with `classmethod` above?

# Function Decoration

- The pattern

```
def f(args): ...  
f = modifier(f)
```

has the following special syntax:

```
@modifier  
def f(args): ...
```

- We can rewrite the previous example to:

## Example

```
class Static:  
    val = 5  
    # class method  
    @classmethod  
    def sqr(c): return c.val * c.val
```

- More examples of using modifiers: Memoisation, Type-checking.

# Memoisation with Function Decorators

- We want a version of Fibonacci (below), that remembers previous results (“**memoisation**”).

## Example

```
def fib(n):  
    """Compute Fibonacci number of @n@."""  
    if n==0 or n==1:  
        return 1  
    else:  
        return fib(n-1)+fib(n-2)
```

- **NB:** This version performs an exponential number of function calls!

# Memoisation with Function Decorators

- To visualise the function calls, we define a decorator for **tracing**:

## Example

```
def trace(f):  
    """Perform tracing on function @func@."""  
  
    def trace_func(n):  
        print("++ computing", f.__name__, " with ", str(n))  
        return f(n)  
  
    return trace_func
```

- and we attach this decorator to our `fib` function:

## Example

```
@trace  
def fib(n): ....
```

# Memoisation with Function Decorators

- Now, we implement memoisation as a decorator.
- **Idea:**
  - ▶ Whenever we call `fib`, we remember input and output.
  - ▶ Before calling a `fib`, we check whether we already have an output.
  - ▶ We use a dictionary `memo_dict`, to store these values.
- This way, we never compute a Fibonacci value twice, and runtime becomes linear, rather than exponential!

# Memoisation with Function Decorators

Here is the implementation of the decorator:

## Example

```
def memoise(f):  
    """Perform memoisation on function @func@."""  
    def memo_func(n, memo_dict=dict()):  
        if n in memo_dict.keys():  
            return memo_dict[n]  
        else:  
            print("++ computing", f.__name__, " with ", str(n))  
            x = f(n)  
            memo_dict[n] = x  
            print(".. keys in memo_dict: ", str(memo_dict.keys()))  
            return x  
    return memo_func
```

# Memoisation with Function Decorators

- We attach this decorator to the `fib` function like this:

## Example

```
@memoise  
def fib(n): ...
```

- Nothing else in the code changes!
- See online sample `memofib.py`

# Interpretation

- Strings can be evaluated using the function `eval`, which evaluates string arguments as Python expressions.

```
>>> x = 5
```

```
>>> eval ("x")
```

```
5
```

```
>>> f = lambda x: eval("x * x")
```

```
>>> f(4)
```

```
16
```

- The command `exec` executes its string argument:

```
>>> exec("print(x+1) ")
```

```
5
```

## Compilation

- This performs compilation of strings to byte-code:

```
>>> c = compile("map(lambda x:x*2,range(10))", # code object
    'pseudo-file.py',          # filename for error msg
    'eval') # or 'exec' (module) or 'single' (stm)
>>> eval(c)
<map object at 0x7f2e990e3d30>
>>> for i in eval(c): print(i)
0 ...
```

- Beware of indentation in the string that you are composing!

```
>>> c2 = compile('''
... def bla(x):
...     print x*x
...     return x
... bla(5)
... ''', 'pseudo', 'exec')
>>> exec c2
25
```

# New features in Python 3.5

Python 3.5 brings several new features, especially:

- Coroutines with `async` and `await` syntax
- A dedicated infix operator for matrix multiplication
- **Type Hints**
- Additional Unpacking Generalizations
- % formatting support for bytes and bytearray
- Pre-3.5 but important:
  - ▶ Several built-in functions now return iterators, rather than lists, e.g
  - ▶ `dict.keys()` and `dict.values()`
  - ▶ builtin higher-order functions such as `map`, `filter`, `reduce`
  - ▶ operators such as `range`
  - ▶ if in doubt, try it in the python shell, e.g.

## Example

```
>>> map(lambda x: x**2 , range(0,10))  
<map object at 0x7f8a87c17978>
```

# Coroutines with async and await syntax

- `async` and `await` are supported, as in C#:

```
import asyncio
async def http_get(domain):
    reader, writer = await asyncio.open_connection(doma
writer.write(b'\r\n'.join([
    b'GET / HTTP/1.1',
    b'Host: %b' % domain.encode('latin-1'),
    b'Connection: close',
    b'', b''
])))
    async for line in reader:
        print('>>>', line)
    writer.close()
loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(http_get('example.com'))
finally:
    loop.close()
```

# A dedicated infix operator for matrix multiplication

- You can use the @ operator for infix matrix multiplication:

## Example

```
res = m1 @ m2
```

- NumPy 1.10 supports this syntax as well:

## Example

```
ones = np.ones(3)
# builds: array([ 1., 1., 1.])

m = np.eye(3)
# builds the unit matrix

res = ones @ m
print(res)
# builds: array([ 1., 1., 1.])
```

# Type Hints

- Type information can be added as hints to function arguments and return values.
- The semantics of these annotations is **undefined**.
- You can't rely on types being checked statically!
- The type `Any` stands for an unknown type.
- Example:

## Example

```
def greeting(name: str) -> str:  
    return 'Hello ' + name
```

# New features in Python 3.6

Python 3.6 brings several new features, especially:

- asynchronous generators
- asynchronous comprehensions
- syntax for variable annotations
- formatted string literals

---

<sup>0</sup>See <https://docs.python.org/3/whatsnew/3.6.html>

# Asynchronous generators

- Python 3.6 adds support for native coroutines and `async / await` syntax to Python 3.5
- This removes a Python 3.5 limitation: not possible to use `await` and `yield` in the same function body;

## Example

```
async def ticker(delay, to):  
    """Yield numbers from 0 to *to* every *delay* seconds  
    for i in range(to):  
        yield i  
        await asyncio.sleep(delay)
```

# Asynchronous generators

- Python 3.6 adds support for native coroutines and `async / await` syntax to Python 3.5
- This removes a Python 3.5 limitation: not possible to use `await` and `yield` in the same function body;

## Example

```
async def ticker(delay, to):  
    """Yield numbers from 0 to *to* every *delay* seconds  
    for i in range(to):  
        yield i  
        await asyncio.sleep(delay)
```

# Asynchronous comprehensions

- Python 3.6 adds support for using `async` for in `list`, `set`, `dict` comprehensions and generator expressions

## Example

```
result = [i async for i in aiter() if i % 2]
```

# Asynchronous comprehensions

- Python 3.6 adds support for using `async` for in `list`, `set`, `dict` comprehensions and generator expressions

## Example

```
result = [i async for i in aiter() if i % 2]
```

# Syntax for variable annotations

## Example

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

captain: str # Note: no initial value!

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

# Formatted string literals

- formatted string literals are similar to the format strings accepted by `str.format()`;
- formatted string literals are prefixed with 'f' and are similar to the format strings accepted by `str.format()`.
- they contain replacement fields surrounded by curly braces

## Example

```
>>> name = "Fred"
>>> f"He said his name is {name}."
'He said his name is Fred.'
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}" # nested f
'result:      12.35'
```

# Formatted string literals

- formatted string literals are similar to the format strings accepted by `str.format()`;
- formatted string literals are prefixed with 'f' and are similar to the format strings accepted by `str.format()`.
- they contain replacement fields surrounded by curly braces

## Example

```
>>> name = "Fred"
>>> f"He said his name is {name}."
'He said his name is Fred.'
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}" # nested f
'result:      12.35'
```