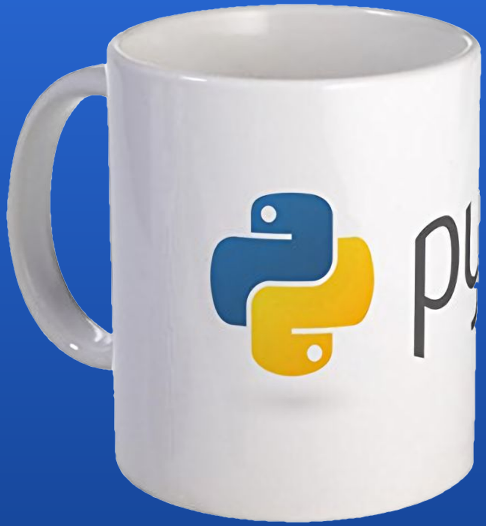
Python Krus



Advanced Python

By Peder Bergebakken Sundt

Programvareverstedet
www.pvv.ntnu.no

self.bio

- Peder Bergebakken Sundt
- 21 years old
- In my third year for a Master of Science in Communication technology
- Worked with Python for ~9 years
- Hangs out on Programvareverkstedet on Stripa on my spare time

This course

- This course with briefly touch upon many cool concepts in higher level Python programming.
- We will mainly use vanilla Python 3 for these slides
- Many of these tricks and methods can be used in Python 2 as well
- Python 3 introduces the new print method, advanced unpacking, parameter annotations and the yield from statement among many other things.
- You’re going to see the character “_” **a lot**.
- Please don’t be afraid to ask if you have any questions or didn’t quite catch something.

The Interactive Interpreter

- The interactive interpreter runs Python code one line at a time.
- Any returned value is printed out, formatted using the `repr()` method
- The code on the left of this slide is how i'll display most of the examples

```
>>> i_return_a_value()
5
>>> 5
5
>>> None
>>> i_return_None()# None is the default return value
>>> 2 + 2
4
>>> "foobar"# return values are printed using repr()
'foobar'
>>> print("foobar")# print() formats using str()
foobar
```

Python is a parsed language

- Python allows dynamic behaviour making the language difficult to compile:

```
>>> print("length:", len("test"))
length: 4
>>> import builtins
>>> setattr(builtins, "len", lambda x: x.__len__() + 5)
>>> print("length:", len("test"))
length: 9
```

- We solve this by running it in an interpreter
- This is the major reason why many believe Python is slow
- This is not always the case, but many use it as a general rule of thumb

The Python parser and interpreter

The execution of Python code is divided into two steps:

1. Parse the source code and compile it into Python bytecode (usually stored in *.pyc files or the __pycache__/ directory)
2. Execute the simplified bytecode in an interpreter (kinda like the Java VM but not really)

This allows for some changes, optimizations and oddities to occur in both stages

Oddities in the Python parser

- Python allows for expressions like

```
if 5 < myFunction() <= 10:
    doSomething()
```

- In a simpler language, `5 < 6 < 7` would be resolved into something like `True < 7`, which is not what we want.
- Python notices a pattern here while parsing the code, and changes the code from `5 < 6 < 7` into `5 < 6 and 6 < 7`
- We can have some fun with this

Example: Some fun with the parser

```
>>> print(5 < 7 < 10)# 5 < 7 and 7 < 10
True
>>> print(2 < 5 > 2)# 2 < 5 and 5 > 2
True
>>> print("a" in "aa" in "aaa")# "a" in "aa" and "aa" in "aaa"
True
>>> print(not 7 == True is not False)# not 7 == True and True is not False
True
```

Variable function arguments

- A Python method can take in a unknown amount of arguments
- These come in the form of lists and dictionaries
- * denotes a list of positional arguments
- ** denotes a list of keyword arguments

```
>>> def myfunc(*args, **kwargs):
...     print(args)
...     print(kwargs)
>>> myfunc(1, 2, 3, 4, foo="bar", five=5)
(1, 2, 3, 4)
{'foo': 'bar', 'five': 5}
```

Advanced unpacking

- Python 2 had iterator unpacking:

```
>>> a, b, c = range(3)
>>> (a, c)
(0, 2)
```

- Python 3 introduces advanced unpacking using similar syntax to *args:

```
>>> a, *rest, b = range(10)
>>> (a, rest, b)
(0, [1, 2, 3, 4, 5, 6, 7, 8], 9)
```

Polymorphism in Python

- Everything in Python is an object (or at least a psuedo object)
  - Functions and classes are objects
  - Even True and False are objects
  - Even the code itself is an object!
- Python 1 introduced function names like __init__() and __str__() to give the different types a common interface:

  `5 == 6` is interpreted as

  `(5).__eq__(6)` by the python parser
- Python uses these methods behind the scenes when running code
- We can overload these!

How to view the contents of an object

```
>>> dir(5)#Lets look at the attributes the object 5 contains
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__',
'__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
'__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
'__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

Type and attribute methods

- Python 1 defined a common interface for objects to implement. This has been built upon and extended since then.
- This convention is what allows us to make our objects able to cooperate as well as they do!
- `if [1, 2]: print("The list has members")`

  is interpreted as

  `if [1, 2].__bool__(): print("The list has members")`

```
myobject.__int__()  == int (myobject)
myobject.__str__()  == str (myobject)
myobject.__repr__() == repr(myobject)
myobject.__bool__() == bool(myobject)
myobject.__len__()  == len (myobject)
myobject.__list__() == list(myobject)
myobject.__iter__() == iter(myobject)
```

Comparison operators

- When you compare two objects, Python needs to know how to compare them.
- A least one of the two objects must implement a comparison method for this to work. This is a method which usually returns either True or False
- `["a", "b"] > None`

  is interpreted as

  `["a", "b"].__gt__(None)`

```
myobject.__lt__(self, other) #Less than
myobject.__le__(self, other) #Less than or equal
myobject.__eq__(self, other) #Equals
myobject.__ne__(self, other) #Not Equal
myobject.__gt__(self, other) #Greater than
myobject.__ge__(self, other) #Greater than or equal
```

Arithmetic operators

- Behaves the same way as comparison operators, except they're not expected to return a boolean
- Right hand side counterparts exists as well
- Operator precedence is handled by the parser and can not be overridden

(as far as i know)

```
object.__add__     (self, other)  ==  self +  other
object.__sub__     (self, other)  ==  self -  other
object.__mul__     (self, other)  ==  self *  other
object.__matmul__  (self, other)  ==  self @  other
object.__truediv__ (self, other)  ==  self /  other
object.__floordiv__(self, other)  ==  self // other
object.__mod__     (self, other)  ==  self %  other
object.__pow__     (self, other)  ==  self ** other
object.__lshift__  (self, other)  ==  self << other
object.__rshift__  (self, other)  ==  self >> other
object.__and__     (self, other)  ==  self &  other
object.__xor__     (self, other)  ==  self ^  other
object.__or__      (self, other)  ==  self |  other
```

Container methods

- Lists, dictionaries, sets, tuples, deques and strings all use the same container interface methods:
- ```
  a = myobject[5]
  myobject["foo"] = "bar"
  del myobject[5]
  ```
  is interpreted as
  ```
  a = myobject.__getitem__(5)
  myobject.__setitem__("foo", "bar")
  myobject.__delitem__(5)
  ```

- Slicing was hacked in as an afterthought:

```
>>> class MyClass:
...     def __getitem__(self, value):
...         print(value)
>>> myobject = MyClass()
>>> myobject[3]
3
>>> myobject[3:4]
slice(3, 4, None)
```
```

Attribute handlers

- All objects must have an implementation of __getattr__, __setattr__ and __delattr__
- Luckily you inherit a very good implementation by default!
- Used whenever you access a member attribute of an object:

```
print(myobject.foo)
```

is executed as

```
print(myobject.__getattr__("foo"))
```

- Similar interface to containers, but must be implemented on all objects

```
>>> class AttributeDict(dict):
...     __getattr__ = dict.__getitem__
...     __setattr__ = dict.__setitem__
...     __delattr__ = dict.__delitem__
>>> mydict = AttributeDict()
>>> mydict["foo"] = 5
>>> print(mydict.foo)
5
```

New style classes and objects

- The concept of a descriptor was introduced late in Python 2.
- In general, a descriptor is an object attribute whose access has been overridden by methods.
- A descriptor is an object with `__get__()`, `__set__()`, and `__delete__()` methods.
- You can easily make these using `property()`
- In Python 2 you had to inherit "object" to get the descriptor logic, while this inheritance is implicit in Python 3.
- Object adds the `__getattribute__`, `__setattribute__` and `__delattribute__` member functions which handle descriptor logic before calling `__getattr__`, `__setattr__` and `__delattr__` respectively.

Properties

```
>>> class MyClass:
...     def foo():
...         doc = "The foo property."
...         def fget(self):
...             return "The value of foo"
...         def fset(self, value):
...             print("foo was set to", value)
...         def fdel(self):
...             pass
...         return locals()
...     foo = property(**foo())
```

```
>>> myobject = MyClass()
>>> myobject.foo = 5
foo was set to 5
>>> print(myobject.foo)
The value of foo
>>> print(MyClass.foo.__doc__)
The foo property.
```

Simpler properties

```
>>> class MyClass:
...     @property
...     def foo(self):
...         return input("What is foo? ")
...     @foo.setter
...     def foo(self, value):
...         print("Foo was set to", value)
...
```

```
>>> myobject = MyClass()
>>> print(myobject.foo)
What is foo? Hello
Hello
>>> print(myobject.foo)
What is foo? World
World
>>> myobject.foo = 5
Foo was set to 5
```

Callables

- An object is a “callable” object if it implements the __call__ method

  ```
  myobject(1, 2)
  ```
  is executed as
  ```
  myobject.__call__(1, 2)
  ```

- **def** handles this for you:

  ```
  >>> def myfunc(): pass
  >>> myfunc.__call__
  <method-wrapper '__call__' of function object at 0x000000E4B2703E18>
  ```

Callable example

```
>>> class Funky:
...     def __call__(self):
...         print("Look at me, I work like a function!")
>>> f = Funky()
>>> f()
Look at me, I work like a function!
```

Lambda functions

- Callables are simply objects
- Because of this we can pass a callable in as an argument to a function
- The lambda statement simplifies this, allowing you to define callables inline:

```
>>> def double(value):
...     return value + value
>>> def call(func):
...     print('func("test") returns:', func("test"))
>>> call(double)
func("test") returns: testtest
>>> call(lambda x: x + x + x)
func("test") returns: testtesttest
>>> call(lambda x: 5)
func("test") returns: 5
```

Class descriptions

- When you define a class in Python, you're in reality storing a callable object, which produces instances of the class you described:
- `MyClass.__call__(*args, **kwargs)`

  is a method which does: *(somewhat simplified)*

```
instance = MyClass.__new__(MyClass, *args, **kwargs) # The instance is constructed by __new__
instance.__init__(*args, **kwargs) # The newly constructed instance is initialized by __init__
return instance
```

Default __new__ constructor simplified

```
class MyClass:
    def __new__(cls, *args, **kwargs):
        self = object() #an empty object
        for attribute_name in dir(cls):
            attribute_value = getattr(cls, attribute)
            if type(attribute_value) is function:
                def instance_method(*args, **kwargs):
                    return attribute_value(self, *args, **kwargs)
                setattr(self, instance_method)
            else:
                setattr(self, attribute_value)
        return self
```

Annotations

- A new feature introduced in Python 3.0, which has not been backported
- Used to annotate what types a function uses and returns

```
>>> def myfunc(a: int, b: str) -> list:
...     assert type(a) is int
...     assert type(b) is str
...     #do something
>>> myfunc.__annotations__
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'list'>}
```

- Python does not enforce these in any way, mainly used for documentation and better assistance from IDEs and linters

Decorators

- Functions are just callable objects
- We can make changes to these callable objects
- This we call "decorating" a function
- A "decorator" is simply a function that takes in a callable object as a parameter and returns the decorated version of that callable object:

```
myfunc = mydecorator(myfunc)
```

Decorator syntax

- Python added syntactical sugar to make this more practical:

```
def myfunc(): pass
myfunc = mydecorator(myfunc)
```

can be written as

```
@mydecorator
def myfunc(): pass
```

- You can stack multiple decorators on a single function

Decorator example: HTML tag

```
>>> def with_b_tag(func):# a decorator
...     def new_func(*args, **kwargs):
...         return "<b>" + func(*args, **kwargs) + "</b>"
...     return new_func
...
>>> @with_b_tag
... def hello_world():
...     return "Hello, World!"
...
>>> print(hello_world())
<b>Hello, World!</b>
```

Decorator example: memoizer

```
>>> def memoize(func):# a decorator
...     memory = {}
...     def new_func(argument):
...         if argument in memory:
...             return memory[argument]
...         else:
...             value = func(argument)
...             memory[argument] = value
...             return value
...     return new_func
...
```

```
>>> @memoize
... def fibonacci(n):
...     if 0 <= n <= 1:
...         return n
...     return fibonacci(n-1) + fibonacci(n-2)
...
>>> print(fibonacci(200))
280571172992510140037611932413038677189525
```

- This saves a lot of runtime

Decorator example: logging

```
>>> def log(func):# a decorator
...     def new_func(*args):
...         print(func.__name__ + str(args))
...         ret = func(*args)
...         print(func.__name__, "returned:", ret)
...         return ret
...     return new_func
...
>>> @log
... def foo(value):
...     return value.upper() + value.lower()
...
```

```
>>> @log
... def bar(value1, value2):
...     return foo(value1)[::-1] + foo(value2)
...
>>> print("final result:", bar("Hello", "World"))
bar('Hello', 'World')
foo('Hello',)
foo returned: HELLOhello
foo('World',)
foo returned: WORLDworld
bar returned: ollehOLLEHWORLDworld
final result: ollehOLLEHWORLDworld
```

Decorators with parameters

- Decorators alone might seem a bit limiting
- Making a decorator for every single edge case is a lot of work
- We can solve this by “cheating” a little
- We can make a function which returns the decorator we want
    - In this course we’ll call them “decorator builders”, but they’re often just called decorators
- This function will be able to take in other parameters as well!

Decorator builder example: Generic HTML tag

```
>>> def with_tag(tag):# a decorator builder
...     def decorator(func):# a decorator
...         def new_func(*args, **kwargs):
...             return "<" + tag + ">" + func(*args, **kwargs) + "</" + tag + ">"
...         return new_func
...     return decorator
...
>>> @with_tag("b")
... @with_tag("i")
... def welcome(name):
...     return "Hello, " + name.split()[0] + "!"
...
>>> print(welcome(input("Enter your name: ")))
Enter your name: Peder B. Sundt
<b><i>Hello, Peder!</i></b>
```

Decorator builder example: with_resource

```
def with_resource(filename):# a decorator builder
    with open(filename, "r") as f:
        file = f.read()

    def decorator(func):# a decorator
        def new_func(*args, **kwargs):
            return func(*args, file, **kwargs)
        return new_func
    return decorator

from flask import Flask# a popular library for web development
import time
app = Flask("My server name")

@app.route("/")
@with_resource("resources/frontpage_template.html")
def frontpage_get(request, template):
    date = time.strftime("%B %d, %Y")
    return template.format({"date": date})
```

Context Managers

```
>>> with open("my_file.txt", "r") as f:
...     data = f.read()
>>> print(data)
I'm awesome!
```

- The `with` statement uses what we call a context manager
- Context managers are simply an object which implements the `__enter__` and `__exit__` methods.
- `__enter__` is called at the start of the `with` block, optionally storing the returned value `as f`.
- `__exit__` is called when exiting the `with` block
- `open()` uses its `__exit__` method to close the file.

Context manager example: HTML Tag

```
>>> class Tag:
...     def __init__(self, tag):
...         self.tag = tag
...     def __enter__(self):
...         print("<" + self.tag + ">")
...     def __exit__(self, type, value, traceback):
...         print("</" + self.tag + ">")
...
>>> with Tag("b"):
...     print("This text is bold!")
<b>
This text is bold!
</b>
```

Context Manager example: Switch Case

```
>>> class switch():
...     def __init__(self, key):
...         self.key = key
...     def __enter__(self):
...         return self.case
...     def __exit__(self, *args):
...         pass
...     def case(self, key):
...         def decorator(func):
...             if self.key == key:
...                 func()
...             return func
...         return decorator
...
```

```
>>> for key in (4, 5, 6):
...     print("key is", key)
...     with switch(key) as case:# the switch
...         @case(4)
...         def unimportant_name():
...             print("foo")
...         @case(5)
...         @case(6)
...         def unimportant_name():
...             print("bar")
...
key is 4
foo
key is 5
bar
key is 6
bar
```

Metaclasses

- Metaclasses can be a controversial topic
- Some believe it overcomplicates the object model
- Whether you want to use them or not is up to you
- They present lots of interesting opportunities for reducing boilerplate and make nicer APIs

What is a Metaclass?

```
>>> class MyClass: pass
>>> type(MyClass)
<class 'type'>
>>> myobject = MyClass()
>>> type(myobject)
<class '__main__.MyClass'>
>>> isinstance(myobject, MyClass)
True
>>> isinstance(MyClass, type)
True
```

- A metaclass is the parent of a class object
- All classes inherit the metaclass `type` by default
- We can therefore make classes using `type` instead of using the `class` statement:

```
>>> MyClass = type('MyClass', (), {})
>>> MyClass
<class '__main__.MyClass'>
```

Using type instead of the class statement

- These two code snippets are (almost) identical:

```
>>> class Foo:
...     x = 5
>>> class Bar(Foo):
...     def get_x(self):
...         return self.x
>>> mybar = Bar()
>>> mybar.get_x()
5
```

```
>>> Foo = type('Foo', (), dict(x=5))

>>> Bar = type('Bar', (Foo,), dict(get_x = lambda self: self.x))


>>> mybar = Bar()
>>> mybar.get_x()
5
```

Metaclasses are callable

- We can use `type` as a function to make classes
- The class statement does the same thing
- This means the class statement should accept any callable as a metaclass

```
>>> class MyClass(metaclass = print):
...     pass
MyClass () {'__module__': '__main__', '__qualname__': 'MyClass'}
>>> print(MyClass)
None
```

Making your own metaclass

- Making your own metaclass is as simple as inheriting `type`:

```
>>> class MyMeta(type):
...     pass
>>> class MyClass1(metaclass = MyMeta):
...     pass
>>> type(MyClass1)
<class '__main__.MyMeta'>
>>> MyMeta("MyClass2", (), {})
<class '__main__.MyClass2'>
```

Iterables

- An terable object is in Python defined as “An object capable of returning its members one at a time.”
- Most of Python considers an object to be iterable if it implements `__iter__`
- Lists, sets, dictionaries, deques, strings and bytearrays among many other implements this interface.
- `__iter__` is a method that returns an Iterator-like object
- The built in function `iter(myobject)` simply returns `myobject.__iter__()`

Iterators

```
>>> myiter = iter([1, 2, 3])
>>> myiter
<listiterator object at 0x7f855c944400>
>>> myiter.next()
1
>>> myiter.next()
2
>>> myiter.next()
3
>>> myiter.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- when we call `myiter.next()` the last time, `StopIteration` is raised instead.
- This is how an iterator signals their end
- This means iterators can have an unknown length

Iterators

- `for` loops will exhaust iterators for you:

```
>>> for i in iter([1, 2, 3]): print(i, end=" ")
1 2 3
```

- `for` loops also call `iter()` for you

```
>>> class MyClass:
...     def __iter__(self): pass
...
>>> for i in MyClass(): print(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: iter() returned non-iterator of type 'NoneType'
```

Generators

- Generators are a kind of iterator which generates its values on-the-fly

- This is achieved by making `iter(mygenerator()).next()` compute the next value when called

- This can save a lot of memory and result in some nifty speedups

- Python 3 changed the `range` method from producing a list to producing a generator:

- Python 2:

```
>>> range(5)
[0, 1, 2, 3, 4]
```

- Python 3:

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
```

The yield statement

- `yield` allows you to make generators with ease
- The `yield` statement resembles `return` in many ways
- When `yield` is called, the value is outputted and the function is halted until next value is requested.
- `return` in a generator will raise a `StopIteration` exception

```
>>> def mygenerator():
...     yield 1
...     print("Hello, World!")
...     yield 2
...     return 3
...
>>> for i in mygenerator(): print(i)
...
1
Hello, World!
2
```

The yield from statement

- The `yield from` was introduced in Python 3.4
- `yield from` is used when you want to pass along the result from an another generator through your own generator
- `yield from` will return any value stored in `StopIteration`

```
>>> def foo():
...     yield 1
...     yield 2
...     return 3
>>> def bar():
...     ret = yield from foo()
...     print("foo returned:", ret)
>>> for i in bar(): print(i)
1
2
foo returned: 3
```

Generator example: execution order

```
>>> def foo():
...     for _ in range(3):
...         yield input("Write something: ")
...     return "I was returned by foo()"
...
>>> def bar():
...     ret = yield from foo()
...     yield ret.upper()
...
```

```
>>> for i in bar():
...     print("I got:", i)
Write something: Alice
I got: Alice
Write something: Bob
I got: Bob
Write something: Foobar
I got: Foobar
I got: I WAS RETURNED BY FOO()
```

AsyncIO

- AsyncIO is a module in the standard library, introduced in Python 3.4
- The syntax was extended in Python 3.5 to make it more intuitive
- It enables you to handle many different input/output streams simultaneously without resorting to threading
- To achieve this, AsyncIO runs a event loop which schedules coroutines to run at different times
- A coroutine is a glorified generator, which yields control back to the event loop while idle

Coroutines

- Coroutines are a language construct designed for concurrent operation.
- They use the halting mechanic of generators to allow for other code to run in the meantime

- Coroutines in Python 3.4:

```
@asyncio.coroutine
def hello_world():
    yield from asyncio.sleep(1)
```

- Python 3.5 added `async` and `await` to simplify this:

```
async def hello_world():
    await asyncio.sleep(1)
```

AsyncIO example: scheduling and concurrency

```
>>> import asyncio
>>> async def coro_1():
...     while True:
...         await asyncio.sleep(1)
...         print("coro_1")
...
>>> async def coro_2():
...     await asyncio.sleep(0.5)
...     while True:
...         await asyncio.sleep(1)
...         print("coro_2")
...
```

```
>>> event_loop = asyncio.get_event_loop()
>>> asyncio.ensure_future(coro_1())
>>> asyncio.ensure_future(coro_2())
>>> event_loop.run_forever()
coro_1
coro_2
coro_1
coro_2
coro_1
coro_2
coro_1
coro_2
```

AsyncIO example: return values

```
>>> import asyncio
>>> async def coro_sub():
...     await asyncio.sleep(1)
...     return 5
...
>>> async def coro_main():
...     ret = await coro_sub()
...     print("coro_sub returned", ret)
...     return 10
...
```

```
>>> event_loop = asyncio.get_event_loop()
>>> event_loop.run_until_complete(coro_main())
coro_sub returned 5
10
```

AsyncIO example: web development

- A real code snippet I've written recently. Using **sanic** as the webserver, **airspeed** as the templating engine and **aiopg** to interact with the database.

```
@app.route("/")
@outputs_html
@with_template("frontpage.vm")
async def GET_frontpage(request, template):
    session = await get_session(request)
    user = await database.get_user(session)
    return template.merge(locals())
```

Why use asyncio?

- It’s new, hip and cool, and built in
- It is way easier to develop and debug than other some of the asynchronous frameworks
- It utilizes the available resources more efficiently than threading
- There is an ever growing library of asyncio modules, capable of cooperating thanks to the common framework

Programvareverkstedet

- It’s at the second floor on Stripa by Adgangskontrollen.
- Need help learning or figuring out something programming related? We’d love to help you out!
- We have a neat server room, computer terminals, a fun community with a great pool of knowledge!
- Open for anyone to just come by, no obligations or duties required!