And now for something completely different

Python

Referenser

- http://docs.python.org/tutorial/introduction.html
- http://docs.python.org/reference/compound_stmts.html
- http://docs.python.org/library/stdtypes.html

Kommentarer

```
# Comment
"""
Multi line comments, also used for documentation purposes
"""
```

Villkor

```
i = 1
if i > 5:
    print("larger")
else:
    print("smaller")

Output:
smaller
```

Loopar

```
for i in range(6, 10):
    print(i)

output:
6
7
8
9
```

Listor

```
# Create a list of animals
s = ["cat", "dog", "giraffe", "ocelot"]
for e in s:
    print(e)

Output:
cat
dog
giraffe
ocelot
```

Listor och tester

```
print( "Do we have a cat? " + str("cat" in s))
print( "Maybe a tiger? " + str("tiger" in s))

s.append("tiger") # I really want a tiger, append one to the list

print( "Now? " + str("tiger" in s))
print("How many do we have? " + str(len(s)) + " " + str(s))

Output:
Do we have a cat? True
Maybe a tiger? False
Now? True
How many do we have? 5 ['cat', 'dog', 'giraffe', 'ocelot', 'tiger']
```

Dictionaries (ordlistor?)

```
d = {
    "Bruce Wayne" : "Batman",
    "Clark Kent" : "Superman",
    "Charlie" : "The Solver"
    }

for k, v in d.iteritems():
        print(k + " is also known as " + v)

Output:
Bruce Wayne is also known as Batman
Charlie is also known as The Solver
Clark Kent is also known as Superman
```

Uppslagning med exceptions

```
try:
    lookup = d["Unknown"]
    print("Found " + lookup)
except KeyError:
    print("Who?")

Output:
Who?
```

Uppslagning utan exceptions

```
something = d.get("Unknown")
if something != None:
    print(something)
else:
    print("No, still nothing")

Output:
No, still nothing
```

Tupler (tuples)

```
t = (1, 2, 3)

def three_args(a, b, c):
    print(a, b, c)
    return c, b, a # we can also return a tuple (multiple return
values!)

r = three_args(2, 3, 4)
print(r)

r = three_args(*t) #expand the tuple into the arguments
print(r)

Output:
(2, 3, 4)
(4, 3, 2)
(1, 2, 3)
(3, 2, 1)
```

Default argument

```
def key_func(foo, bar = "nah"): #bar has a default value
    print(foo + " " + bar)

key_func("hum") #will use the default value
key_func( bar = "wrong order", foo = "thats right") # argument by name

d = { "foo": "fooz", "bar":"barz" }
key_func(**d) # expand and map the arguments from the dictionary

Output:
hum nah
thats right wrong order
fooz barz
```

Slump

```
funky = range(10)
print("what is this? " + str(funky))
random.shuffle(funky)
print("shuffled? " + str(funky))
print("random value: " + str(random.randint(20, 30)))

Output:
what is this? [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
shuffled? [8, 6, 4, 5, 3, 9, 7, 0, 1, 2]
random value: 24
```

Skivor (slices)

```
s = "abcdef"
print(s[0]) # a
print(s[3:]) # def
print(s[:3]) # abc

#Also works on lists

Output:
a
def
abc
```

Långa strängar

```
s = """
Some really long message
On multiple lines
But with no real meaning
"""

i = 1
for c in s.splitlines():
    if len(c.strip()) > 0:
        print(str(i) + " "+ c)
    i=i+1

Output:
2 Some really long message
3 On multiple lines
4 But with no real meaning
```

Långa strängar #2

```
s = """
Some really long message
On multiple lines
But with no real meaning
"""

for i, c in enumerate(s.splitlines()):
    if len(c.strip()) > 0:
        print(str(i+1) + " " +  c)

Output:
2 Some really long message
3 On multiple lines
4 But with no real meaning
```

Klasser

```
class Foo:
        """Example class to show basic principles"""

        i = 17
        #private things begin with __
        __name = "default"

        def __init__(self, name):
            print("A new Foo is born: " + name)
            self.__name = name

        def Print(self):
            print(self.__name + " " + str(self.i))

f = Foo("Foo1")

f.__name = "Fooz" # will not change the private data
f.i = 33 # But this is still valid
f.Print()

Output:
A new Foo is born: Foo1
Foo1 33
```

Inte skrivet i sten

```
# You can still add new things to the instance,
# if its good practice is another question..
f.fnord = 23

def Forgotten(self):
    print("oops, forgot to add a method")

Foo.Forgotten = Forgotten
f.Forgotten()

Output:
oops, forgot to add a method
```

Arv

```
class Bar(Foo):
    def __init__(self, name):
        print("A new Bar is born: " + name)
        self.__name = name
        self.i = 11

b = Bar("Bar1")
b.Print() #WTF?! Why doesn't it set the name correctly?

Output:
A new Bar is born: Bar1
default 11

# __ means class private data
# _ normally indicates private data, but has no real implications
```

Monster!

```
class Monster:
    looks = "scary"
    def __init__(self):
        self.strength = random.randint(10,20)
        self.hp = random.randint(50,60)

    def __str__(self):
        return "Monster with: " + str(self.strength) + " " +
str(self.hp)

monsters = [ Monster() for i in range(10) ]

for m in monsters:
    print(m)

Output:
Monster with: 20 52
Monster with: 13 53
Monster with: 18 55
Monster with: 18 54
# prints 10 monsters
```

Iteratorer

```
class MyIterator:
    def __init__(self, s):
        self.i = 0
        self.s = s

    def __iter__(self):
        return self

    def next(self):
        if self.i >= len(self.s):
            raise StopIteration
        self.i = self.i + 1
        return self.s[:self.i] #slice of the i:th last chars

it = MyIterator("abcde")
for i in it:
    print(i)

Output:
a
ab
abc
abcd
abcde
```

Iteratorer 2

```
class MonsterContainer:
    def __init__(self):
        # Create some new monsters
        self.monsters = [ Monster() for i in range(5)]

    def __iter__(self):
        return iter(self.monsters) # return a iterator over our
monsters

mc = MonsterContainer()
for m in mc:
    print(m)

Output:
Monster with: 10 51
Monster with: 14 59
Monster with: 16 55
Monster with: 18 53
Monster with: 14 60
```

Generatorer

```
# yield means it returns a value,
# pauses execution and will continue there
# when the iterator calls next() again

# We want to generate an infinite number of confusing replies
def Generator():
    while True:
        yield random.choice(["Yeah?", "Hum..", "You sure?", "Really?",
"I'm confused"])

for i, q in enumerate(Generator()):
    if i >= 5:
        break
    print(q)

Output:
You sure?
Hum..
Yeah?
Yeah?
Really?
```

Multipla yields

```
# because we continue where we left off, we can have multiple yield if
we wanted to
def RepeatAfterMe():
    while True:
        yield "Socks"
        yield "Pants"
        yield "Shoes"
        yield "Not the other way around"

for i, q in enumerate(RepeatAfterMe()):
    if i > 8:
        break
    print(str(i) + " " + q)

Output:
0 Socks
1 Pants
2 Shoes
3 Not the other way around
4 Socks
5 Pants
6 Shoes
7 Not the other way around
8 Socks
```

Yield

```
def fromTo(fr, to):
    while fr <= to:
        yield fr
        fr += 1

for i in fromTo(10, 13):
    print(i)


Output:
10
11
12
13
```

Yield again

```
def printEm(iterable):
    for i in iterable:
        print(i)

printEm(i * i for i in fromTo(15, 20))


Output:
225
256
289
324
361
400
```

Decorators