

Advanced Python Concepts

Scientific Programming with Python

Nicola Chiapolini

University of Zurich
Faculty of Science

17 July 2026

Control Flow

else on Loops

- ▶ for- and while-loops can have an else-clause
- ▶ executed if loop terminated normally (i.e. no break)

```
for n in range(2, 10):  
    for x in range(2, n):  
        if n % x == 0:  
            print(n, '=', x, '*', n//x)  
            break  
    else:  
        print(n, 'is a prime number')
```

2 is a prime number
3 is a prime number
4 = 2 * 2
5 is a prime number
6 = 2 * 3
7 is a prime number
8 = 2 * 4
9 = 3 * 3

- ▶ it's often possible and cleaner to refactor a loop to not need this at all (especially in functions as you can use return instead of break)

match

- ▶ similar to switch from other languages
- ▶ but does pattern-matching, variable assignment and more

```
match coord:
    case (0, 0):                # a constant
        print("Origin")
    case (0, d) | (d, 0):       # two patterns, assign to d
        print(f"on an axis, {d} from origin")
    case (x, 0):                # will never match due to (d, 0) above
        print(f"X-axis at {x}")
    case (x, y) if x == y:      # matches only if guard is true
        print("diagonal")
    case (x, y):                # match any 2-tuple
        print("at", coord)     # remember coord :-)
    case _:                     # matches everything
        raise ValueError("Not a point")
```

DEMO

match: case Patterns

- ▶ `_` matches everything
- ▶ can be simple constants
- ▶ can contain variables that will get assigned
- ▶ `|` combines alternatives (or)
- ▶ tuple and list patterns work like unpacking
- ▶ dict patterns match keys present (additional keys are ignored)
- ▶ can match instances of classes and their properties

```
case (x, 0, *_): | {"x": x, "y": 0} | Point(x=x, y=0):  
    print(f"On X-Axis at {x} (or lost in space)")
```

Additionally case can contain a guard: works as in list-comprehension

Exceptions

Exceptions

- ▶ stop the execution and alert calling code
- ▶ typical use-case: something wrong with argument passed to function
- ▶ calling code can catch exception, e.g. to recover or log
- ▶ common **built-in exceptions**
 - TypeError** object has the wrong type, e.g. `int` instead of `list`
 - ValueError** value of the object is wrong, e.g. negative but should be positive
 - IndexError** index is not available, e.g. in `list` or `tuple`
 - KeyError** key is not available, e.g. in a `dict`
 - LookupError** `IndexError` or `KeyError` (plus edge-cases)

Complex Example

```
def read(n, container):  
    while True:  
        try:  
            container.append(  
                int(input(">>> "))  
            )  
            n -= 1  
        except ValueError:  
            print("Input ignored.")  
            continue  
        except TypeError:  
            print("Press CTRL+D to stop")  
            continue  
        except EOFError:  
            break  
    if n < 1:  
        break
```

ValueError

user input is no int

TypeError

n can not be decremented

EOFError

user pressed CTRL+D

AttributeError

container has no append

KeyboardInterrupt

user pressed CTRL+C

DEMO

Catching Exceptions

uses `try ... except` block

```
try:
    container.append(
        int(input(">>> "))
    )
    n -= 1
except ValueError:
    print("Input ignored.")
    continue
except TypeError:
    print("Press CTRL+D to stop")
    continue
except EOFError:
    break
```

- ▶ if exception is raised in `try` block, go to next `except`
- ▶ test if raised exception-type is subclass of specified type
 - yes: execute block, ignore other `except`
 - no: go to next `except`
- ▶ if no further `except`: propagate exception to calling code
- ▶ *Note*: complex `try ... except` block are often due to bad design

Advanced Catching

```
def read(n, container):
    while True:
        try:
            container.append(
                int(input(">>> "))
            )
            n -= 1
        except (ValueError, TypeError):
            continue
        except: # NOT recommended
            break
        else:
            print(n, "left to enter")
    finally:
        print("current:", container)
    if n < 1:
        break
```

`except (ValueError, TypeError):`
catch ValueError or TypeError

`except:`
catch *ANY* exception
really *NOT* recommended!

`else:`
execute code if no exception is raised
(unrelated to `except:` above)

`finally:`
specify code that is guaranteed
to be executed.

DEMO

Guaranteed Execution

- ▶ `finally` is executed before handling any `return`, `break` or `continue` in the rest of the block.
- ▶ `finally` can prevent these from happening

```
def pitfall():  
    try:  
        print("entered try")  
        return "try"  
    finally:  
        print("entered finally")  
        return "finally"  
  
print("returned", pitfall())
```

```
entered try  
entered finally  
returned finally
```

- ▶ putting `return`, `break` or `continue` into `finally` triggers a `SyntaxWarning` from Python 3.14 on.

Raising Exceptions Manually

- ▶ using `raise` statement

- ▶ after checking some condition

```
if not isinstance(container, (list, deque)):
    raise TypeError("container must be list or deque")
```

- ▶ after catching an exception

```
except AttributeError as e:
    raise TypeError("container is not appendable")
```

- ▶ raising the class works as well, e.g. `raise TypeError`
- ▶ but recommended to use instance with helpful message
- ▶ passing more than one argument is allowed

```
raise TypeError("container must be list or deque", type(container))
TypeError: ('container must be list or deque', <class 'dict'>)
```

User-defined Exceptions

- ▶ built-in exceptions form a class-hierarchy
 - ▶ e.g. `IndexError` and `KeyError` are subclasses of `LookupError`

- ▶ own exception types can be added to allow precise signals
 - ▶ use specific subclasses where applicable

```
class NegativeValueError(ValueError):  
    pass  
  
raise NegativeValueError("Be positive.")
```

- ▶ inherit from `Exception` when unsure

```
class FittingError(Exception):  
    pass  
  
raise FittingError("Fit did not converge.")
```

- ▶ *Note:* user-defined exceptions should not inherit from `BaseException`

Storing

`except ... as ...`: allows to assign the error to a variable

- ▶ to extract arguments

```
try:
    raise TypeError("container must be list or deque", type(container))
except TypeError as e:
    message, type_ = e.args
    # more error handling
```

- ▶ to investigate type

```
except LookupError as err:
    if isinstance(err, KeyError):
        pass # imagine a special case
    pass # imagine lot of common code
```

- ▶ for chaining

Chaining

New exceptions can be raised while already handling an exception

- ▶ re-raise current exception

```
try:
    raise RuntimeError("This is bad")
except RuntimeError as err:
    print("We have an error:", err)
    raise
```

- ▶ new exception with implicit chain (same behaviour if bug in error handler)

```
raise TypeError("RuntimeErrors are not my type.")
```

- ▶ new exception with explicit chain (from err)

```
raise TypeError("RuntimeErrors are not my type.") from err
```

- ▶ new exception suppressing chain (from None)

```
raise TypeError("RuntimeErrors are not my type.") from None
```

Python sets

Python sets

- ▶ lesser known sibling of famous containers (lists, dictionaries and tuples)
- ▶ act like a mathematical set
- ▶ **unordered** and **cannot contain duplicates**.
- ▶ entries must be hashable
- ▶ created with `set()` or with `{}` (c.f. dicts)

```
dict_ = {1: "a", 2: "b"} # key-value pairs create a dict
set1 = {1, 2, 2, "a"}    # single values create a set
set2 = set([2, "a", "b", 4, 4])
print("set1:", set1)
print("set2:", set2)
```

```
set1: {1, 2, 'a'}
set2: {'b', 2, 'a', 4}
```

In-Place Operations

► add with `add()`

```
set1.add(3)
set1.add(3)
```

```
set1: {3, 1, 2, 'a'}
```

► remove with `remove()`

```
set1.remove(3)
```

```
set1: {1, 2, 'a'}
```

► add multiple elements with `update()`

```
set1.update([2, 4, 5, 6])
```

```
set1: {1, 2, 'a', 4, 5, 6}
```

Mathematical Operations

```
set1 = {1,2,3,4,5,6,7,8}
```

```
set2 = {1,3,5,7,9,11,13}
```

► union()

```
set1.union(set2) # set1 | set2
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13}
```

► intersection()

```
set1.intersection(set2) # set1 & set2
```

```
{1, 3, 5, 7}
```

► difference()

```
set1.difference(set2) # set1 - set2
```

```
{8, 2, 4, 6}
```

► symmetric_difference() (xor)

```
set1.symmetric_difference(set2)  
# set1 ^ set2
```

```
{2, 4, 6, 8, 9, 11, 13}
```

Tests

▶ element in set

```
3 in {1, 2, 4, 8} False
```

▶ issubset()

```
{1,2}.issubset({1,2,3,4}) # <= True
```

▶ proper subset

```
{1,2} < {1,2} False
```

▶ issuperset()

```
{1,2,3,4}.issuperset({1,2}) # >= True
```

▶ isdisjoint()

```
{1,2}.isdisjoint({4,3}) True
```

DEMO

Aside: Hashable vs. Immutable

- ▶ distinct but related concepts
- ▶ dict-keys and set-values required *hashable* objects
- ▶ a hash is a unique fingerprint for an object and must not change

immutable

- ▶ constant once created
e.g. `no obj[0] = ...`
- ▶ e.g. numbers, strings, tuples
- ▶ are usually also hashable

hashable

- ▶ hash of object is defined
i.e. can call `hash(obj)`
- ▶ e.g. numbers, strings, some tuples, many custom objects

Aside: Hashable vs. Immutable

- ▶ distinct but related concepts
- ▶ dict-keys and set-values required *hashable* objects
- ▶ a hash is a unique fingerprint for an object and must not change

immutable

- ▶ constant once created
e.g. `no obj[0] = ...`
- ▶ e.g. numbers, strings, tuples
- ▶ are usually also hashable

hashable

- ▶ hash of object is defined
i.e. can call `hash(obj)`
- ▶ e.g. numbers, strings,
some tuples,
many custom objects

Aside: Hashable vs. Immutable – Tuples

- ▶ tuples are immutable, so should be hashable
- ▶ however elements of tuples do not need to be immutable
- ▶ if an element is not hashable, the tuple is not hashable

```
t1 = (1,2,3)
print("hash(t1):", hash(t1))
print("hash((1,2,3)):", hash((1,2,3)))
```

```
list_ = [1,2]
t2 = ("a",list_)
list_[0] = 3 # modifies t2
print("t2:", t2)
try:
    hash(t2)
except TypeError as e:
    print("TypeError:", e)
```

```
hash(t1):          529344067295497451
hash((1,2,3)): 529344067295497451
t2: ('a', [3, 2])
TypeError: unhashable type: 'list'
```

Aside: Hashable vs. Immutable – Hashable Objects

- ▶ Objects are hashable by default
- ▶ Hash is based on `id()`
- ▶ `f != g` unless `f is g`

```
class Foo:                                1st: <Foo(1), 8763584230158>
    def __init__(self, val):              2nd: <Foo(2), 8763584230158>
        self.val = val                    False
    def __repr__(self):
        return f"<Foo({self.val}), {hash(self)}>"

f = Foo(1)
print("1st:", f)
f.val = 2
print("2nd:", f)
g = Foo(2); print(f == g)
```

Aside: Hashable vs. Immutable – Unhashable Objects

- ▶ defining a custom `__eq__` makes object unhashable

```
class Foo:                                     3rd: TypeError: unhashable type: 'Foo'
    def __init__(self, val):
        self.val = val
    def __eq__(self, other):
        return self.val==other.val
    def __repr__(self):
        return f"<Foo({self.val}), {hash(self)}>"

f = Foo(1)
try:
    print("3rd:", f)
except TypeError as e:
    print("TypeError:", e)
```

Aside: Hashable vs. Immutable – Evil Objects

```
class Foo:
    def __init__(self, val):
        self.val = val
    def __eq__(self, other):
        return self.val==other.val
    def __hash__(self):
        return self.val
    def __repr__(self):
        return f"<Foo({self.val}), {hash(self)}>"
```

1st: <Foo(1), 1>
2nd: <Foo(2), 2>
3rd: <Foo(2), 2>
4th: <Foo(2), 2>

```
f = Foo(1)
g = Foo(2)
print("1st:", f)
print("2nd:", g)
f.val = 2
print("3rd:", f)
print("4th:", g)
```

Aside: Hashable vs. Immutable – Broken Sets

```
f = Foo(1)
g = Foo(2)
s = {f,g}
print("1st:", s)
f.val = 2
print("2nd:", s)
s.remove(f)
print("3rd:", s)
print("f in s:", f in s)
print("g in s:", g in s)
f.val=3
print("4th:", s)
```

1st: {<Foo(1), 1>, <Foo(2), 2>}

2nd: {<Foo(2), 2>, <Foo(2), 2>}

3rd: {<Foo(2), 2>}

f in s: False

g in s: False

4th: {<Foo(3), 3>}

- „If a class defines mutable objects and implements an `__eq__()` method, it should not implement `__hash__()`“ ([Python Reference](#))

Packing/Unpacking

Packing/Unpacking

packing

- ▶ combine values into tuple
- ▶ no parenthesis needed
- ▶ e.g. returning multiple values

```
tuple_ = 2,3,5,7  
print("tuple_:", tuple_)  
tuple_: (2, 3, 5, 7)
```

unpacking

- ▶ turn elements of sequence into individual variables
- ▶ not just tuples

```
a, b, c, d = tuple_  
print("c:", c)  
c: 5
```

The *-Operator

The *-Operator has two useful uses:

- ▶ specify exactly one “catch-all”-name when unpacking
- ▶ to force unpacking of one or more iterables

extended iterable unpacking

```
a, b, *c = [1, 2, 3, 4]
print("a:", a)
print("b:", b)
print("c:", c)

a: 1
b: 2
c: [3, 4]
```

- ▶ c can end up as an empty list

starred expression

```
unified = *[1,2], *[3,4]
print("unified:", unified)

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

DEMO

Functions

*args and **kwargs

- ▶ `*args` works like the examples for packing/unpacking above.
- ▶ `**kwargs` works very similar, but
 - ▶ exists only in parameter-lists of functions
 - ▶ packs/unpacks keyword-arguments into/from a dictionary

```
def func(*args, **kwargs):  
    print('args:', args)  
    print('kwargs:', kwargs)  
  
func(42, label="answer")  
  
myargs = [1, 2, 3, 4, 5]  
mykwargs = {'foo': 11, 'bar': 13}  
func(*myargs, **mykwargs)
```

```
args: (42,)  
kwargs: {'label': 'answer'}  
args: (1, 2, 3, 4, 5)  
kwargs: {'foo': 11, 'bar': 13}
```

Parameter/Argument Types

Do not confuse the different parameter/argument types

function definition

required parameters specified without a default value: `def f(p):`

optional parameters have a default value: `def f(p=42):`

calling a function

positional arguments are given without a keyword `f(0)`

key-word arguments are passed using a keyword `f(p=0)`

Both required and optional parameters can be passed as positional or key-word arguments!

But Wait...

```
>>> f(p=0)
```

```
TypeError: f() got some positional-only arguments passed as keyword arguments: '0'
```

Forcing Argument Types

You can force the argument type when defining a function:

```
def fun(pos, /, free, *, kw):  
    print("pos: ", pos)  
    print("free:", free)  
    print("kw:  ", kw)  
  
fun(1, 2, kw=3)  
fun("a", free="b", kw="c")
```

pos:	1
free:	2
kw:	3
pos:	a
free:	b
kw:	c

before / only positional arguments allowed

after * only keyword arguments allowed

```
fun(0, 0, 0)
```

```
TypeError: fun() takes 2 positional arguments but 3 were given
```

Forcing Argument Types

You can force the argument type when defining a function:

```
def fun(pos, /, free, *, kw):  
    print("pos: ", pos)  
    print("free:", free)  
    print("kw:  ", kw)  
  
fun(1, 2, kw=3)  
fun("a", free="b", kw="c")
```

pos:	1
free:	2
kw:	3
pos:	a
free:	b
kw:	c

before / only positional arguments allowed

after * only keyword arguments allowed

```
fun(0, 0, 0)
```

```
TypeError: fun() takes 2 positional arguments but 3 were given
```

Lambda Expressions

- ▶ creates anonymous functions
- ▶ gives you a nameless function object

```
lambda a, b: a**2 + b**2
```

```
def fun(a,b):  
    return a**2 + b**2
```

- ▶ useful when passing a helper-function as an argument

```
input_ = [  
    (1, 6),  
    (3, 2),  
    (0, 4),  
]  
  
input_.sort(key=lambda x: x[1] - x[0])  
print("sorted:", input_)
```

```
sorted: [(3, 2), (0, 4), (1, 6)]
```

Factories

Sometimes we want to create a function programmatically:

- ▶ instead of a function that returns, e.g. an integer (an object), a tuple (an object) or a custom object
- ▶ we create a function that returns a function (an object)

Example: Create a function $f(n)$ that returns a function $p(x) = x^n$

DEMO

Factories

Sometimes we want to create a function programmatically:

- ▶ instead of a function that returns, e.g. an integer (an object), a tuple (an object) or a custom object
- ▶ we create a function that returns a function (an object)

```
def make_power_func(power):  
    def func(x):  
        return x ** power  
  
    return func  
  
pow3 = make_power_func(3)  
print("2**3:", pow3(2))
```

2**3: 8

Factories

Sometimes we want to create a function programmatically:

- ▶ instead of a function that returns, e.g. an integer (an object), a tuple (an object) or a custom object
- ▶ we create a function that returns a function (an object)

```
def make_power_func(power):  
    return lambda x: x ** power
```

2**3: 8
2**4: 16

```
pow3 = make_power_func(3)  
print("2**3:", pow3(2))
```

```
pow4 = make_power_func(4)  
print("2**4:", pow4(2))
```

Factories: Looping

Now let's create the whole family in a loop:

```
def make_power_funcs(max_power):  
    pf = []  
    for power in range(max_power+1):  
        pf.append(lambda x: x**power)  
  
    return pf  
  
funcs = make_power_funcs(4)
```

Factories: Looping

Now let's create the whole family in a loop:

```
def make_power_funcs(max_power):  
    pf = []  
    for power in range(max_power+1):  
        pf.append(lambda x: x**power)  
  
    return pf  
  
funcs = make_power_funcs(4)  
print("2**3:", funcs[3](2))  
print("2**1:", funcs[1](2))
```

2**3: 16
2**1: 16

What?!

Factories: Looping

Now let's create the whole family in a loop:

```
def make_power_funcs(max_power):  
    pf = []  
    for power in range(max_power+1):  
        pf.append(lambda x: x**power)  
  
    return pf
```

```
funcs = make_power_funcs(4)  
print("2**3:", funcs[3](2))  
print("2**1:", funcs[1](2))  
print("[3]: ", id(funcs[3]))  
print("[1]: ", id(funcs[1]))
```

```
2**3: 16  
2**1: 16  
[3]: 139879154038592  
[1]: 139879154038112
```

These really are different functions!

Factories: Lookup

- ▶ problem is the binding/lookup of the values
- ▶ first look at a normal function with a global variable
- ▶ then move the same function into a factory

```
def make_power_funcs(max_power):  
    pf = []  
    for power in range(max_power+1):  
        pf.append(lambda x: x**power)  
  
    return pf
```

```
2**3: 16  
2**1: 16
```

```
funcs = make_power_funcs(4)  
print("2**3:", funcs[3](2))  
print("2**1:", funcs[1](2))
```

Factories: Lookup

- ▶ problem is the binding/lookup of the values
- ▶ first look at a normal function with a global variable
- ▶ then move the same function into a factory

```
greeting = "Hello"
```

```
Dear World
```

```
def greet(name):  
    print(greeting, name)
```

```
greeting = "Dear"
```

```
greet("World")
```

Factories: Lookup

- ▶ problem is the binding/lookup of the values
- ▶ first look at a normal function with a global variable
- ▶ then move the same function into a factory

```
def make_greet():  
    greeting = "Hello"  
    def func(name):  
        print(greeting, name)  
  
    greeting = "Dear"  
    return func  
  
greet = make_greet()  
greet("World")
```

Dear World

Factories: Lookup

- ▶ problem is the binding/lookup of the values
- ▶ first look at a normal function with a global variable
- ▶ then move the same function into a factory

```
def make_greet(greeting):  
    def func(name):  
        print(greeting, name)  
  
    greeting = "Dear"  
    return func  
  
greet = make_greet("Hello")  
greet("World")
```

Dear World

The returned function is bound to the factory's **final scope**!

Factories: Lookup

- ▶ problem is the binding/lookup of the values
- ▶ first look at a normal function with a global variable
- ▶ then move the same function into a factory

```
def make_power_funcs(max_power):  
    pf = []  
    for power in range(max_power+1):  
        pf.append(lambda x: x**power)  
  
    return pf
```

```
2**3: 16  
2**1: 16
```

```
funcs = make_power_funcs(4)  
print("2**3:", funcs[3](2))  
print("2**1:", funcs[1](2))
```

The final scope contains `power=4`

Factories: Looping Solutions

- ▶ use default arguments (evaluated when function is defined)
- ▶ use `functools.partial`

```
def make_power_funcs(max_power):  
    pf = []  
    for power in range(max_power+1):  
        pf.append(lambda x,p=power: x**p)  
  
    return pf  
  
funcs = make_power_funcs(4)  
print("2**3:", funcs[3](2))  
print("2**1:", funcs[1](2))
```

2**3: 8
2**1: 2

Factories: Looping Solutions

- ▶ use default arguments (evaluated when function is defined)
- ▶ use `functools.partial`

```
from functools import partial

def exp(x,p):
    return x**p

def make_power_funcs(max_power):
    pf = []
    for power in range(max_power+1):
        pf.append(
            partial(exp, p=power)
        )

    return pf
```

```
2**3: 8
```

```
2**1: 2
```

functools.partial

- useful whenever you want to fix arguments of an existing function
e.g. builtin `pow(base, exp)`

```
def bind_power(power):  
    def func(x):  
        return pow(x, power)  
    return func
```

```
pow3 = bind_power(3)  
print("2**3:", pow3(2))
```

2**3: 8

```
from functools import partial  
pow3 = partial(pow, exp=3)  
print("2**3:", pow3(2))
```

2**3: 8

Wrapping Functions

Sometimes we want to run code before/after every call to a function

- ▶ factory-function takes target-function as argument
- ▶ factory-function creates helper-function
- ▶ helper-function runs our code then calls target-function

Example: Log all calls to a function (e.g. `sorted`)

DEMO

Wrapping Functions

Sometimes we want to run code before/after every call to a function

- ▶ factory-function (`watch`) takes target-function (`func`) as argument
- ▶ factory-function creates helper-function (`wrapped_func`)
- ▶ helper-function runs our code then calls target-function

```
def watch(func):  
    def wrapped_func(*args, **kwargs):  
        print("called", func.__name__)  
        print("  args:", args)  
        print("  kw:  ", kwargs)  
        return func(*args, **kwargs)  
    return wrapped_func
```

Wrapping Functions

Sometimes we want to run code before/after every call to a function

- ▶ factory-function (`watch`) takes target-function (`func`→`sorted`) as argument
- ▶ factory-function creates helper-function (`wrapped_func`→`sorted_wrapped`)
- ▶ helper-function runs our code then calls target-function

```
def watch(func):  
    def wrapped_func(*args, **kwargs):  
        print("called", func.__name__)  
        print("  args:", args)  
        print("  kw:  ", kwargs)  
        return func(*args, **kwargs)  
    return wrapped_func
```

```
called sorted  
  args: ([2, 3, 1, 4, 5],)  
  kw:   {'reverse': True}  
[5, 4, 3, 2, 1]
```

```
sorted_watched = watch(sorted)  
print(sorted_watched([2,3,1,4,5], reverse=True))
```

Wrapping Functions: Built-Ins

- ▶ we can assign our wrapped function to the original name
- ▶ to restore the original function, we can use the `builtins`-module

```
def watch(func):
    def wrapped_func(*args, **kwargs):
        print("called", func.__name__)
        print("  args:", args)
        print("  kw:  ", kwargs)
        return func(*args, **kwargs)
    return wrapped_func

import builtins
sorted = watch(sorted)
print("watch:", sorted([2,3,1,4,5], reverse=True))
sorted = builtins.sorted # restore original function
print("orig: ", sorted([2,3,1,4,5], reverse=True))
```

```
called sorted
  args: ([2, 3, 1, 4, 5],)
  kw:   {'reverse': True}
watch: [5, 4, 3, 2, 1]
orig:  [5, 4, 3, 2, 1]
```

Wrapping Functions: Decorators

- ▶ Decorators are syntactical sugar.
- ▶ They apply a wrapping function to a function definition.

```
@watch
def countdown(start):
    """ count down from start """
    print(start)
    if start > 1:
        countdown(start-1)

countdown(3)
```

```
called countdown
  args: (3,)
  kw:  {}
3
called countdown
  args: (2,)
  kw:  {}
2
called countdown
  args: (1,)
  kw:  {}
1
```

functools.wraps

- ▶ useful to preserve function information

```
help(countdown)
```

```
wrapped_func(*args, **kwargs)
```

functools.wraps

- useful to preserve function information

```
from functools import wraps
```

```
def watch(func):  
    @wraps(func)  
    def wrapped_func(*args, **kwargs):  
        print("called", func.__name__)  
        return func(*args, **kwargs)  
    return wrapped_func
```

```
@watch
```

```
def countdown(start):  
    """ count down from start """  
    print(start)  
    if start > 1:  
        countdown(start-1)
```

```
help(countdown)
```

```
countdown(start)  
    count down from start
```

Context Manager

Context Manager

Manage the runtime context of a `with` block

- ▶ object that has `__enter__` and `__exit__` methods
- ▶ `__enter__` may return an object
- ▶ `__exit__` is guaranteed to be executed (cleanup)
- ▶ most well-known: `open()`

```
with Context() as var:  
    # do something
```

translates to

```
# execute context entering code  
var = result_from_context_entering_code  
# do something  
# execute context leaving code
```

DEMO

Creating a Context Manager

`__enter__` download and open a remote file

`__exit__` close and delete the file

```
class RemoteFile:
    def __init__(self, path, url):
        self._path = path
        self._url = url
        self._file = None

    def __enter__(self):
        subprocess.run(["wcurl", "-o", self._path, self._url], check=True)
        self._file = open(self._path, "r")
        return self._file

    def __exit__(self, type_, value, traceback):
        self._file.close()
        os.remove(self._path)
```

Using Our Context Manager

```
exists = os.path.exists
local = "/tmp/crbn-data"
remote = "https://crbn.ch/data"

print("before", exists(local))
with RemoteFile(local, remote) as file:
    print(file.readline()[:10])
    print("during", exists(local))
print("after ", exists(local))
```

```
before False
7,0,2,0,5,
during True
after False
```

- ▶ `RemoteFile(...)` creates an object
- ▶ `with` calls `__enter__` on it
- ▶ `as` assigns return value

```
manager = RemoteFile(local, remote)
with manager as file:
```

Using Our Context Manager

```
exists = os.path.exists
local = "/tmp/crbn-data"
remote = "https://crbn.ch/data"

print("before", exists(local))
with RemoteFile(local, remote) as file:
    print(file.readline()[:10])
    print("during", exists(local))
print("after ", exists(local))
```

```
before False
7,0,2,0,5,
during True
after False
```

- ▶ `RemoteFile(...)` creates an object
- ▶ `with` calls `__enter__` on it
- ▶ `as` assigns return value

```
manager = RemoteFile(local, remote)
with manager as file:
```

Using Our Context Manager

```
exists = os.path.exists
local = "/tmp/crbn-data"
remote = "https://crbn.ch/data"

print("before", exists(local))
with RemoteFile(local, remote) as file:
    print(file.readline()[:10])
    print("during", exists(local))
print("after ", exists(local))
```

```
before False
7,0,2,0,5,
during True
after False
```

- ▶ `RemoteFile(...)` creates an object
- ▶ `with` calls `__enter__` on it
- ▶ `as` assigns return value

```
manager = RemoteFile(local, remote)
with manager as file:
```

Iterators & Generators

Iterators

Object representing a stream of data

- ▶ can be used in for-loops or `...in ...`
- ▶ must implement `__iter__` and `__next__` methods
- ▶ typically provided by containers (e.g. `my_list = [1,2,3]`)

```
print("loop:")  
for i in my_list:  
    print(i)
```

```
loop:  
1  
2  
3
```

```
print("dunder:")  
it = my_list.__iter__()  
print(it.__next__())  
print(it.__next__())  
print(it.__next__())
```

```
dunder:  
1  
2  
3
```

- ▶ built-in functions `iter()` and `next()` provide access to dunder-methods

Iterators

Object representing a stream of data

- ▶ can be used in for-loops or `...in ...`
- ▶ must implement `__iter__` and `__next__` methods
- ▶ typically provided by containers (e.g. `my_list = [1,2,3]`)

```
print("loop:")  
for i in my_list:  
    print(i)
```

```
loop:  
1  
2  
3
```

```
print("built-in:")  
it = iter(my_list)  
print(next(it))  
print(next(it))  
print(next(it))
```

```
built-in:  
1  
2  
3
```

- ▶ built-in functions `iter()` and `next()` provide access to dunder-methods

Iterators: Stopping

- ▶ no more elements is signaled with a `StopIteration`-Error
- ▶ a `for`-loops handles this correctly

```
print("built-in:")
it = iter(my_list)
print(next(it))
print(next(it))
print(next(it))
try:
    print(next(it))
except StopIteration:
    print("done")
```

```
built-in:
1
2
3
done
```

Iterators: `__iter__` Twice

- ▶ containers and iterators have an `__iter__`-method

```
it = iter(my_list)
print(it)
print(iter(it))
```

```
<list_iterator object at 0x7fa9a21beef0>
<list_iterator object at 0x7fa9a21beef0>
```

- ▶ only iterators have a `__next__`-method

```
try:
    print(next(my_list))
except TypeError as e:
    print("failed:", e)
```

```
failed: 'list' object is not an iterator
```

allows to pass container or iterator to `for`-loop

Iterators: `__iter__` Differences

`container.__iter__` returns new iterator (freshly initialized)

`iterator.__iter__` returns `self` (can be exhausted already)

```
print("container")
for i in my_list:
    print("a", i)
print("--")
for i in my_list:
    print("b", i)
print("--")
```

```
container
a 1
a 2
a 3
--
b 1
b 2
b 3
--
```

```
print("iterator")
it = iter(my_list)
for i in it:
    print("a", i)
print("--")
for i in it:
    print("b", i)
print("--")
```

```
iterator
a 1
a 2
a 3
--
--
```

Iterators: Defining an Object

```
class Squares:
    def __init__(self, start=0, num=100):
        self._current = start
        self._limit = start+num

    def __iter__(self):
        return self

    def __next__(self):
        if self._current >= self._limit:
            raise StopIteration
        val = self._current**2
        self._current += 1
        return val
```

```
for i in Squares(num=5):
    print(i)
```

```
0
1
4
9
16
```

DEMO

Generators

First approximation: alternative way to create iterator

- ▶ function with at least one `yield` statement

```
def squares(start=0, num=100):  
    i = start  
    while i < start+num:  
        yield i**2  
        i += 1
```

0
1
4
9
16

```
for i in squares(num=5):  
    print(i)
```

- ▶ calling the function just creates an iterator (no code executed)
- ▶ calling `next()` on iterator runs code (until next `yield`)
- ▶ if function returns a `StopIteration` error is raised instead

Generators: Advanced

- ▶ can receive new values with `send()`

```
def squares(start=0, num=100):  
    i = start  
    while i < start+num:  
        new = yield i**2  
        if new == None:  
            i += 1  
        else:  
            i = new  
  
it = squares(num=5)  
print(next(it))  
print(it.send(3))  
print(next(it))
```

0
9
16

All Together Now

```
import builtins
import contextlib

@contextlib.contextmanager
def prefix(pre):
    def new_print(*args, **kwargs):
        orig_print(pre, *args, **kwargs)
    orig_print = builtins.print
    builtins.print = new_print
    yield orig_print
    builtins.print = orig_print

with prefix("NB:") as noprefix:
    noprefix("no prefix here")
    print("we're inside")
print("and out again")
```

no prefix here
NB: we're inside
and out again