Python advanced class

Module 4, Functions, lambdas, and decorators

- Varargs and kwargs
- Lambda functions, closures, and functions as parameters
- Decorators
- Generator functions

Varargs and kwargs

Positional arguments and Keyword arguments

- When calling a function it is possible to use positional arguments and keyword arguments
- Positional arguments does not indicate the parameter they adhere to, they just follow the position of the parameter
- Keyword arguments are prefix with the parameter they adhere to and may be written in any order
- Positional arguments must come before any keyword argument

In [1]:
```
def transfer(from_account, to_account, amount, expedite=False, printed_
    pass

transfer(1234, 4567, amount=200_000, printed_receipt=True)
```

Varargs

- If a function should work with any number of positional arguments, it needs a varargs parameter
- A varargs parameter will contain all remaining positional arguments in a list
- The varargs parameter will have a `*` infront of it

In [2]:
```python
def rms(*args):
    count = len(args)
    s = sum(x ** 2 for x in args) / count
    return s ** 0.5

print(rms(1, 2, 3, 2))
```

```
2.1213203435596424
```

Kwargs

- Likewise for functions that should be able to receive any number of keyword arguments
- The kwargs parameter will contain all remaining keyword arguments in a dictionary
- The kwargs parameter will have a `**` infront of it

In [3]:
```python
def contributions(**contributors):
    names = []
    s = 0
    for name, amount in contributors.items():
        names.append(name)
        s += amount
    print(f"The people: {', '.join(names)} has given {s}")

contributions(andrew=10, ben=20, charlie=35)
```

```
The people: andrew, ben, charlie has given 65
```

Using lists and dicts as parameters

- If the arguments to a function is given in a list or a dictionary the opposite is also possible
- The argument is now prefixed with either `*` or `**`

In [4]:
```
def pyt(a, b):
    return (a ** 2 + b ** 2) ** 0.5

args = [3.0, 4.0]
kw = {'a': 6.0, 'b': 8.0}
print(pyt(*args))
print(pyt(**kw))
```

```
5.0
10.0
```

Lambda functions, closures, and functions as parameters

Functions as objects

- A function is an object like anything else in Python
- When a function is defined it becomes a name in the symboltable bound to a *callable* object
- It is possible to make a list of functions, have a function take a function as a parameter, and/or return a function
- The docstring is an attribute of the function

Lambda function

- A lambda expression is an expression that evaluate to an anonymous function object
- A lambda expression can only have a single expression, not for loops or compound statements
- A lambda expression is written as `lambda` *parameter(s)* `:` *expression*

In [5]:
```
f = lambda x, y: x/y + y/x
print(f(8, 5))
```

```
2.225
```

Closure variables

- The expression may contain values that belong to variables that does not exist any more
- It is possible to write functions that return functions based on the parameters

In [6]:

```python
def mk_compare(precision=1):
    return lambda a, b: round(a, precision) == round(b, precision)

cmp3 = mk_compare(3)
cmp5 = mk_compare(5)
print(cmp3(3.12, 3.13))
print(cmp3(4.5573, 4.5571))
print(cmp5(3.12, 3.13))
print(cmp5(4.5573, 4.5571))
```

```
False
True
False
False
```

Sorting

- When sorting a list we need a way to compare two values for which one is the largest
- With a lambda function we can transform both values to something comparable (int or str)
- This lambda function is passed as the `key=` argument

In [7]:

```python
names = ["Dennis", "Andrew", "Irving", "eric", "ben", "Charlie", "fred"
print(sorted(names))
print(sorted(names, key=lambda name: name.lower()))
print(sorted(names, key=lambda name: len(name)))
```

```
['Andrew', 'Charlie', 'Dennis', 'George', 'Irving', 'ben', 'eri
c', 'fred', 'hector', 'john']
['Andrew', 'ben', 'Charlie', 'Dennis', 'eric', 'fred', 'George',
'hector', 'Irving', 'john']
['ben', 'eric', 'fred', 'john', 'Dennis', 'Andrew', 'Irving', 'h
ector', 'George', 'Charlie']
```

Decorators

What is a decorator?

- A decorator is a function that wraps another function to do something before or after a function call
- The purpose can be to transform, validate, or repeat something in the function call
- A decorator can make it simple to do something implicit that would otherwise require extra steps

Using a decorator

- a decorator is written immediately before a function definition with an `@` infront

In [8]:
```python
from functools import cache

@cache
def pyt(a, b):
    print(f"I was called with {a} and {b}")
    return (a ** 2 + b ** 2) ** 0.5

print(pyt(3.0, 4.0))
print(pyt(3.0, 4.0))
print(pyt(6.0, 8.0))
```

```
I was called with 3.0 and 4.0
5.0
5.0
I was called with 6.0 and 8.0
10.0
```

Defining decorators

- The easiest way to make a decorator is to use the `functool.wraps`

In [9]:
```python
from functools import wraps

def log_calls(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        print(f"Log: {f.__name__} is called with {args}, {kwargs}")
        r = f(*args, **kwargs)
        print(f"Log: {f.__name__} returned {r}")
        return r
    return wrapper

@log_calls
def pyt(a, b):
    return (a ** 2 + b ** 2) ** 0.5

print(pyt(3.0, 4.0))
```

```
Log: pyt is called with (3.0, 4.0), {}
Log: pyt returned 5.0
5.0
```

Generator functions

How does a for loop work?

- An iterable type is a type that can be iterated over in a for loop
- When a for loop is executed the iterable must provide a method called `__iter__` that returns an iterator object
- The iterator object must provide a method called `__next__` that can return the next value in the iteration
- When the iterator object has reached the end of the iterable, it must raise a special exception called `StopIteration`

In [10]:
```python
L1 = [1, 2, 3]
it = L1.__iter__()
print(it.__next__())
print(it.__next__())
print(it.__next__())
try:
    print(it.__next__())
except StopIteration:
    print("No more values")
```

```
1
2
3
No more values
```

The keyword yield

- It is definitely possible to create a class that implements this interface
- But often it is easier to create a generator function
- A generator function returns a new value each time it reaches a yield line

In [11]:
```
def repeat_string(n, string):
    for _ in range(n):
        yield string

for text in repeat_string(3, "Hello"):
    print(text)
```

```
Hello
Hello
Hello
```

Using a generator function

- A generator function will not return a value
- It returns an iterator object that can fetch all values one at a time
- If a generator function must return a finish dataset it can be wrapped in a list constructor

In [12]:
```
def repeat_string(n, string):
    for _ in range(n):
        yield string

L1 = list(repeat_string(3, "Hello"))
print(L1)
```

```
['Hello', 'Hello', 'Hello']
```