CM3-Computer Science

Our Sessions

Lecture	Big Idea
1. What is the essence of a tree?	Approach a problem at the right level of abstraction
2. Functions can travel first class too.	Idempotent functions and Functional Programming
3. Do not change things to make them faster.	Unchangedness and Immutability of data
4. Your state of mind is not my state of mind.	Modeling the state of objects with Object-Oriented Programming
5. If it looks and quacks like a duck, it is a duck.	Polymorphism and well defined interfaces.
6. If they cant eat bread, let them eat trees.	How the choice of the right data structure leads to performant algorithms.
7. Any problem can be solved by adding one extra layer of indirection	Separate the specification of an algorithm from its implementation
8. You may not use the bathroom unless I give you the key!	Concurrency and Parallelism in programs
9. Floating down a lazy river.	Lazy Evaluation and Streams
10. Deforming Mona Lisa with a new tongue.	Languages for stratified design
11. It’s turtles all the way down!	Languages and the machines which run them
12. Feed me Seymour, and I’ll remember you!	GPUs enable the learning instead of the writing of programs
13. Hey Markov! Whats the next word?	Large Language Models generate the next token.


1. Whst is the essance of a tree?
3. the right lev of tsesm
2. Aporgechee things to
7 rrat lse or dase? . .
5 Opcideeren ing ttablam to
5. raling ttesepraslfiol dostcation
8 Idempront fuctcries
8 Utchamange is to hat
7, perrenidide prfacre.
8. Your robs chamg sres
7 nenicl-odfinrd alliaiscions.
9 Mowl the bedsts an of icking
1. m abking my ecia preferattions.
1. Oolting the calsk lo foschiing
7 be ceiling sazngle eres duk.
1 vikk a porfast eade.
2. Appombent fuctions
5. Unchaggdantt treke is nott
5 make ct dinrellulakticle. .
10. Hmy problemn can solved
6 anckking lefia lifractes. .
3. Dlempotrert futrees
4 Uneaianef fincions
8 Polanynlom
5 rrif its me tate l of ducks
8. Plyocinste a a duck
1 Umphonem and docks
1 and ir a dees
9 Undaichoct is 'ana olind
7 fak rats orfjtten duck.
10 Parchlangle or bucs.
7 lfs the olse, opiueles.

Python Function Scope

Variables defined *locally* are not available outside.

```
def f(a, b):
    return a + b, a - b
f(1,2)
print(a)
```

Output:

```
NameError: name 'a' is not defined
```

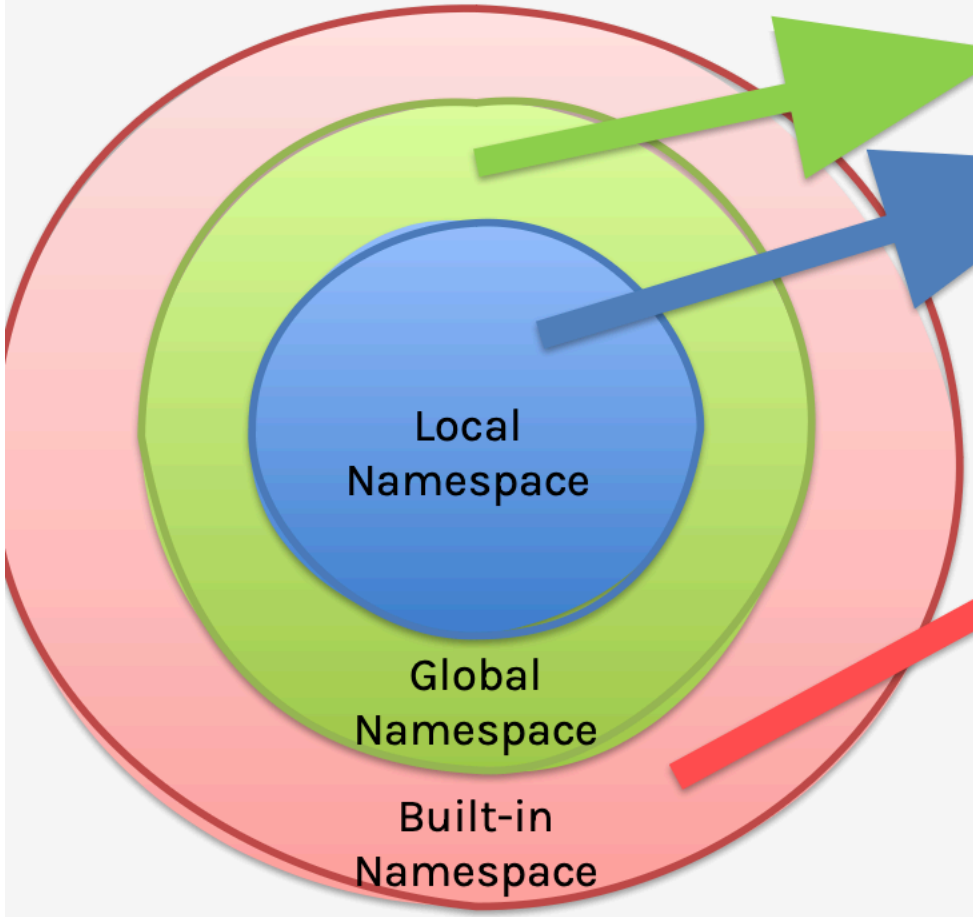
Variables defined in loops (including loop index) are available after.

```
for m in range(2):
    print(m)
print(m)
```

gives us:

```
0
1
1
```

Namespaces




```
def square(x):
    newvar = x**2
    return newvar
square_list = list()
for numbe      range(10):
    squar_    st.append(square(number))
squa    e_list
>>>
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

First Class Functions

Functions can be assigned to variables: `hypot = lambda x, y : sqrt(x*x + y*y)`.

Functions can also be passed to functions and returned from functions.

Functions passed:

```
def mapit(aseq, func):
    return [func(e) for e in aseq]
mapit(range(3), lambda x : x*x) # [0, 1, 4]
```

map and reduce are famous built-in functions.

Functions returned:

```
def soa(f): # sum anything
    def h(x, y):
        return f(x) + f(y)
    return h
sos = soa(lambda x: x*x)
sos(3, 4) # returns 25 like before
```

Closures: capturing state

Sometimes we want to capture some state:

```
def soaplusbias(f, bias): # sum anything
    def h(x, y):
        return f(x) + f(y) + bias
    return h
sosplusbias = soaplusbias(lambda x: x*x, 5)
sosplusbias(3, 4)
```

Returns 30

The last line is indentical to before, but we have additionally captured a `bias` from the enclosing scope. The bias is captured when we define `sosplusbias`, but gets used when we call it. This is called a **closure**. It is useful at all sorts of places where state must be captured and used later, as in deep learning callbacks, GUIs, etc

Stack Frames

- Pure functions have referential transparency: you can at any time substitute the call to the function by the value it produces. The answer is the same all the time and there are no side effects like some dictionary being populated
- Functions have local environments called frames, and these are put on a stack, which is a last-in-first-out data structure
- Pure functions cannot represent something that is time-dependent. We will need to create state variables in our environment. We can do this using our stack frames using closures.

A better way: objects

- But there is a better way to do this: the notion of an object as a dictionary, or repository of state. It is the same idea as a stack frame, but is attached to an object
- Now you can track the life-cycle of an object.
- In fact, in Python, everything is an object: integers, floats, everything.
- just like there are functions in an environment, there are methods on objects
- just like each function gets its own stack frame, each object has its own state variable and its own notion of time.

Classes

In Python, a class is simply defined as:

```
class MyClass:
    pass
```

which is equivalent to saying

```
class MyClass(object):
    pass
```

```
class Curry:
    def set_info(self, name, desc):
        self.name = name
        self.desc = desc

    def print(self):
        print(f'{self.name} curry,
              {self.desc} is Yummy!')
```

attribute method self

Mutating State

```
class BankAccount:

    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        self.balance = self.balance - amount
```

We make an *instance* of a *class* BankAccount called myaccount with a balance of 100 and withdraw 20 from it. Both the "function" withdraw and the variable representing the balance are called as if they belong to the instance myaccount.

```
myaccount = BankAccount(100)
print(myaccount.balance) # 100
myaccount.withdraw(20)
print(myaccount.balance) # 80
```

- `__init__`: a constructor for the class. This is the function called when we say `BankAccount(100)`.
- Why does `__init__(self, balance)` have 2 arguments then? This is because it is a very special kind of function called a **method**, in which the first argument is the *instance* of the class. By convention, it is always called `self` in python. Thus we will use `BankAccount(balance)` to call the *constructor method*.
- `withdraw(self, amount)`: another *method*. Once again `self` is the existing account object. You can think of your program's `myaccount` instance `withdraw`-ing the `amount` and thus write it `myaccount.withdraw(amount)`, the implicit `self` having been moved to the left of the dot.
- `myaccount.balance`: this is an *instance variable*, some data 'belonging" to the instance, just as the previous method did. Thus we'll use `self.balance` inside the methods to denote it.

Class Variables and Class Methods

How do we share variables and functionality across all instances of a class?

```
class BankAccount:
    max_balance = 1000

    @classmethod
    def make_account(cls, balance):
        if balance <= cls.max_balance:
            return cls(balance)
        else:
            raise ValueError(f"{balance} too large")

    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        self.balance = self.balance - amount

ba = BankAccount.make_account(100)
ba.balance # 100
ba.withdraw(20)
ba.balance # 80

bb = BankAccount.make_account(10000) # ValueError: 10000 too large
```

- `max_balance` is a **class variable** since this max value needs to be shared by all accounts. Note it is not preceded with a `self`.
- The **classmethod** `make_account`, announced thus by *decorating* it with `@classmethod`, takes the class, NOT the instance, as its "implicit" (as in not written) first argument, moving the class over to the left side of the dot.
- It calls the constructor as `cls(balance)` if the class variable `cls.max_balance` is reasonable.

Pseudo-code is an abstraction for real code. Lets see some real code!

Lets see even more python

https://github.com/hult-cm3-rahul/LearningPython