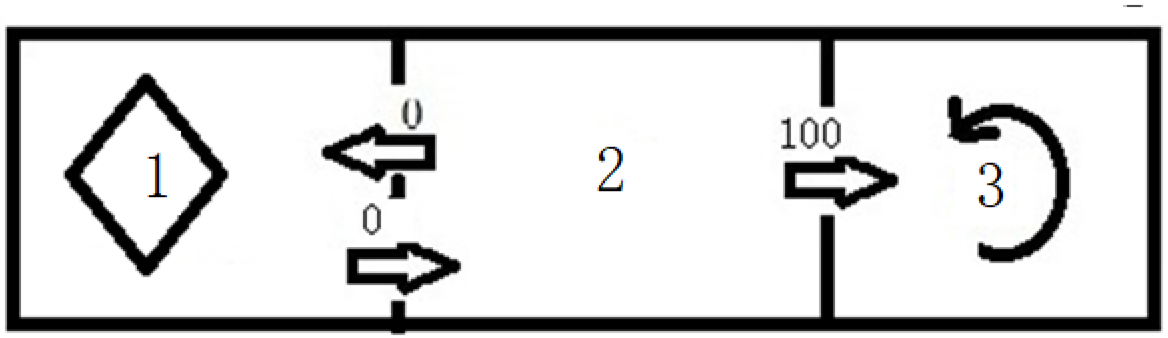
<CAPS>

Session 5 – Implementing Value Iteration in Python

Leo Klenner, Henry Fung, Cory Combs

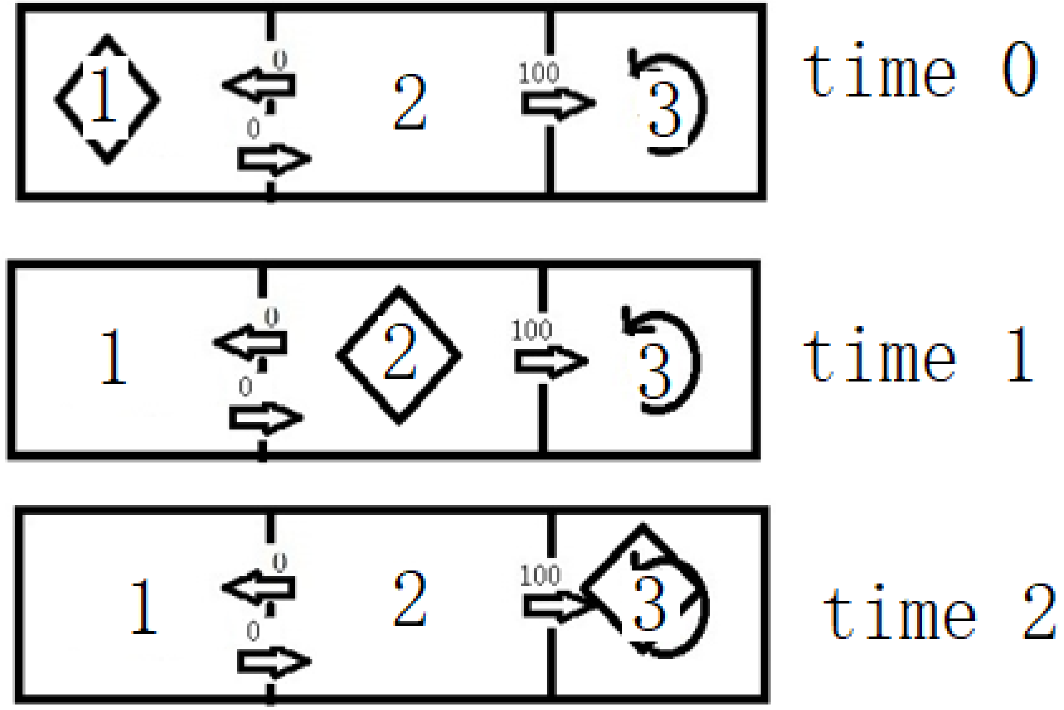
A Simpler Grid World I

- > Consider a simple three grid world
- > The agent has 3 states (grids 1, 2, 3), and two action choices (forward f and backward b)
- > Goal: Find shortest route to 3
- > At time 0, robot observe its current state ($s_0 = 1$) chooses action ($a_0 = f$), transitions to grid 2 (s'= s_1 = 2), and receives immediate reward ($r_1 = 0$)



A Simpler Grid World II

- At time 1, robot observe its current state ($s_1 = 2$) chooses action ($a_1 = f$), transitions to grid 3 (s'= s_2 = 3), and receives immediate reward ($r_2 = 100$)
- Episode ends
- Reward as a function of current state, action, and next state:
- R= f(s,a,s')
  - R(1,f,2)?
  - R(2,f,3)?
  - R(3,b,2)?
  - R(2,b,1)?



The Stochastic Matrix

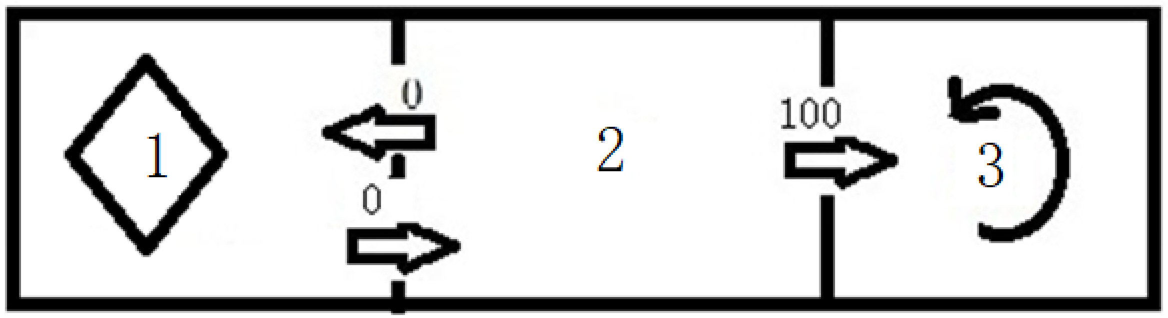
> What if state transition is not deterministic?
>
> > I start in A, I move forward but might get stuck in the mud and remain in A. What if state transition is not deterministic?

> We can represent the probabilities of moving from state i (or, given that you are in i) to state j in one time period as: $\Pr(j|i) = P_{ij}$

$$P = \begin{bmatrix} P_{1,1} & P_{1,2} & \dots & P_{1,j} & \dots & P_{1,S} \\ P_{2,1} & P_{2,2} & \dots & P_{2,j} & \dots & P_{2,S} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{i,1} & P_{i,2} & \dots & P_{i,j} & \dots & P_{i,S} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{S,1} & P_{S,2} & \dots & P_{S,j} & \dots & P_{S,S} \end{bmatrix}$$

The State Transition Function

> Another representation: P(s,a,s')
> > P(1,f,2) = 0.8
> > P(1,f,1) = 0.2 (stuck in mud)
> > P(1,f,3) = 0
>
> > P(2,f,3) = 1
>
> > P(3,f,3) = 1 (robot in the absorbing state)
> > P(3,b,2) = 0



Policy Iteration

> An algorithm that iteratively evaluate and improve its policy π.

> Do until π’ converges:

1. Choose an arbitrary policy π’
2. Evaluate current policy by solving a system of equations (one eq-s for each state):

$$
\begin{aligned}
V_\pi(s) &= \mathrm{E}\left[R(s,\pi(s),s') + \gamma V(s')\right] \\
&= \sum_{s' \in \mathcal{S}} T(s,\pi(s),s')\left[R(s,\pi(s),s') + \gamma V(s')\right], \qquad \forall s \in \mathcal{S}
\end{aligned}
$$

3. Improve policy at each state (choose action that maximize expected reward, if baseline policy is used thereafter

$$\pi'(s) \leftarrow \arg\max_a \left(E[r \mid s,a] + \gamma \sum_{s' \in S} P(s' \mid s,a) V^\pi(s')\right)$$

State Vector v

```
action_array[action] = np.sum(np.multiply(u, np.dot(v, T[:,:,action])))
```

```
v = np.array([[0.0, 0.0, 0.0, 0.0,
               0.0, 0.0, 0.0, 0.0,
               1.0, 0.0, 0.0, 0.0]])
```

```
### starting state = (1,1)
```

Transition Matrix T

```
action_array[action] = np.sum(np.multiply(u, np.dot(v, T[:,:,action])))
```

T[:][:][0]

```
# action=UP

# state s, s'

# s'   (1,3)(2,3)(3,3)(4,3)(1,2)(2,2)(3,2)(4,2)(1,1)(2,1)(3,1)(4,1)   # s
array([[0.9, 0.1, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ], ###(1,3)
       [0.1, 0.8, 0.1, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ], ###(2,3)
       [0. , 0.1, 0.8, 0.1, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ], ###(3,3)
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ], ###(4,3)
       [0.8, 0. , 0. , 0. , 0.2, 0. , 0. , 0. , 0. , 0. , 0. , 0. ], ###(1,2)
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ], ###(2,2)
       [0. , 0. , 0.8, 0. , 0. , 0. , 0.1, 0.1, 0. , 0. , 0. , 0. ], ###(3,2)
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ], ###(4,2)
       [0. , 0. , 0. , 0. , 0.8, 0. , 0. , 0. , 0.1, 0.1, 0. , 0. ], ###(1,1)
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.1, 0.8, 0.1, 0. ], ###(2,1)
       [0. , 0. , 0. , 0. , 0. , 0. , 0.8, 0. , 0. , 0.1, 0. , 0.1], ###(3,1)
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.8, 0. , 0. , 0.1, 0.1]]) ###(4,1)
```

Python Basics

a. Strings and numbers
b. Variables
c. Conditionals
d. Iteration
e. Data structures
f. Decomposition and abstraction
g. Functions
h. Recursion
i. Modules, packages, libraries

Strings

- Strings are data types that store sequences of characters
  - “cooperate” -> “cooperate”
  - print(“cooperate”) -> cooperate
  - print(“cooperate” + “defect”) -> cooperatedefect
  - 3 * print(“defect”) -> defectdefectdefect
  - type(“defect”) -> <class 'str’>
  - len(“cooperate”) -> 9

  - # this is not a string but a comment

Numbers

- Numbers are data types that store numeric values
  - 3 + 5 -> 8
  - 5 – 3 -> 2
  - 5 / 3 -> 1.6666666666666667
  - 5 // 3 -> 1
  - 3 * 5 -> 15
  - 3 ** 5 -> 243
  - type(3) -> <class ‘int’>
  - type(1.6666666666666667) -> <class ‘float’>
  - abs(-2) -> 2

Working with Strings and Numbers

- Strings are mutable (we can change their individual elements)
- Numbers are immutable (we cannot change their individual elements)

- Operations on strings
  - `“cooperate”[0] -> “c”`
  - `“cooperate”[8] -> “e”`
  - `“cooperate”[-1] -> “e”`
  - `“cooperate”[0:3] -> “coo”`
  - `“cooperate”[:-1] -> “cooperat”`
  - `“cooperate”[::-1] “etarepooc”`
  - `“cooperate”.index(“c”) -> 0`
  - `“cooperate”.count(“o”) -> 2`

Variables

- Variables point to values, Python uses "=" to establish reference
  - `cooperate = 3`
  - `print(cooperate) -> 3`
  - `defect = 5`
  - `print(defect) -> 5`
  - `print(cooperate + defect) -> 8`
  - `cooperate = defect`
  - `print(cooperate) -> 5`
  - `defect += 10`
  - `print(defect) -> 15`

Special Variables – Booleans and None

- There are two Booleans values, `True` and `False`. Python uses `==` to compare variables or values
  - `3 == 3 -> True`
  - `3 == 5 -> False`
  - `type(3 == 3) -> <class 'bool'>`
  - Other comparing operators are `>, <, >=, <=, !=`

- Python's version of nothing is `None`
  - `player_one = None`
  - `player_one -> # empty`
  - `print(player_one) -> None`

Conditionals – if

- We need conditionals to execute code depending on a Boolean condition, i.e. whether a specific relation is `True` or `False`
- Python has three conditionals `if, else, elif`
- `if`
  - `cooperate = 3`
  - `if cooperate == 3:` condition

    ```
        print(cooperate) -> 3
    ```

    consequent (note the indentation)
  - `defect = 5`

    ```
    if defect == 3:
        print(defect) -> # as defect != 3, consequent will not be executed
    ```

Conditionals – else

- We need `else` to epress the alternative of the condition in our `if` statement
- `if + else`
  - `cooperate = 3`
  - `if cooperate > 3:`

```
        print("I'll cooperate")
    else:
        print("I'll defect") -> I'll defect
```

takes no condition

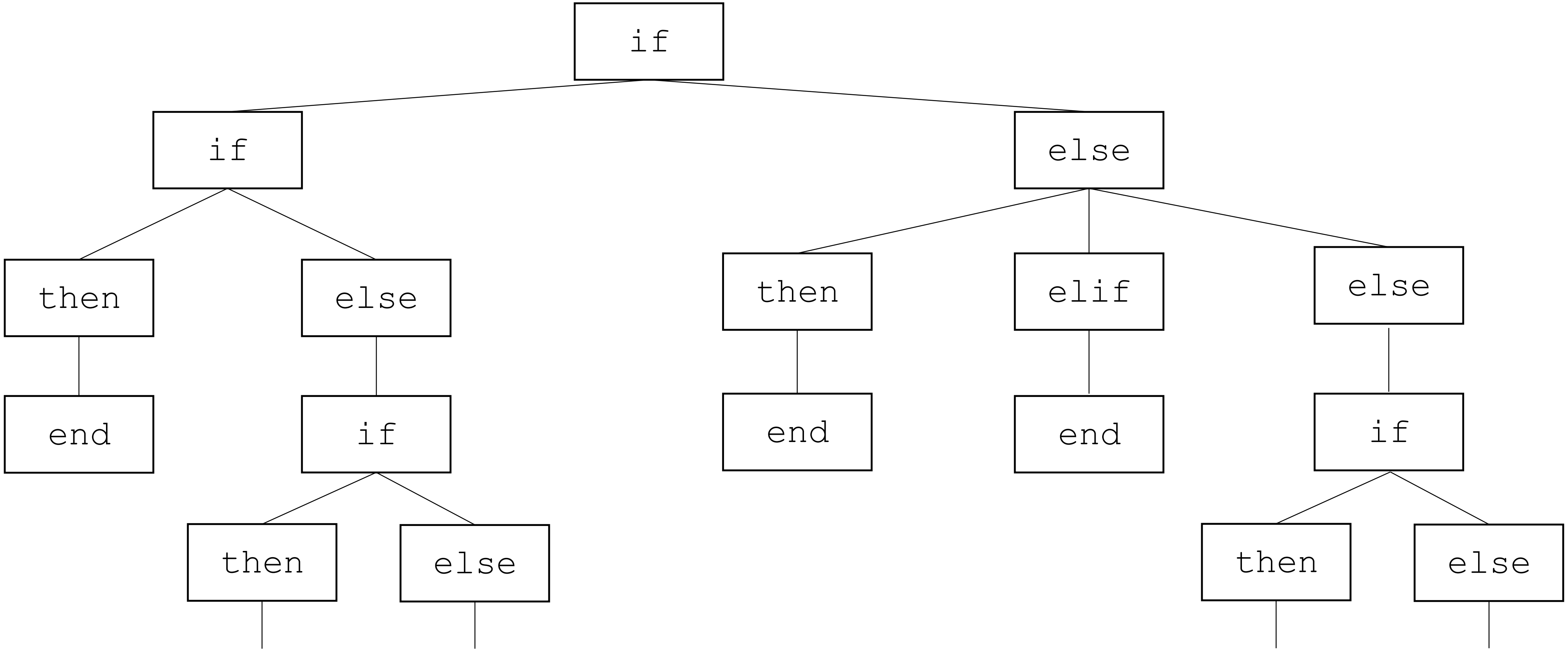
Conditionals – elif

- So far we have only checked for one condition. With `elif` we can check for one than one condition
- if + else + elif
  - `cooperate = 0`
  - `if cooperate > 3:` → condition A

```
    print("I'll cooperate")
elif cooperate == 0:          → condition B
    print("I'll stop playing") -> I'll stop playing
else:                         → alternative of A and B
    print("I'll defect")
```

Working with Conditionals

- Conditionals can be nested and are useful tools for breaking down problems
- Conditionals are the core parts of branching algorithms



Iteration

- Execute set of instructions n times where n > 1
- End when condition is reached
- Two ways to provide this condition, `while` and `for`

- while loop
  - ```
    while <condition>:
        <expression>
        <expression>
        …
    ```
  - `<condition>` evaluates to a Boolean
  - if `<condition>` is `True`, do all the steps insude the `while` code block
  - check `<condition>` again
  - end if `<condition>` is `False`

- for loop
  - ```
    for <variable> in range(<some_num>):
        <expression>
        <expression>
        …
    ```
  - each time through the loop, `<variable>` takes a value
  - first time, `<variable>` starts at the smallet value
  - next time, `<variable>` gets the prev value +1
  - etc.

Pseudo-Application of Iteration to Pathfinding

```
> while position != reward:
      if right == clear:
          move = right
      elif right == blocked:
          move = forward
      elif right == blocked \
      and front == blocked:
          move = left
      else:
          move = back
```

```
> for step in range(len(longest_path):
      if right == clear:
          move = right
      elif right == blocked:
          move = forward
      elif right == blocked \
      and front == blocked:
          move = left
      else:
          move = back
      if position == reward:
          break
```

if final `step` < `len(longest_path)`

Iteration – Assumptions

- Do `while` and `for` require different information about an environment?
- Are there scenarios in which you can use one but no the other?
- Which of `while` and `for` is more general than the other?
- Would you use `while` or `for` to solve the pathfinding problem?

Data Structures

- Wegner and Reily (2003)
  - "A data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to them"

- Data structures enable efficient management, access and manipulation of data
- Different data structures enable different forms of management and manipulation
- In Python, we focus on lists and dictionaries (and ignore tuples and sets)

Data Structures – Lists

- Lists are ordered sequences of elements
  - `payoffs  = [3, 0, 5, 1]`
- Lists are mutable
  - `payoffs[0] -> 3`
  - `payoffs[0] = 99`
  - `print(payoffs) -> [99, 0, 5, 1]`
  - `payoffs.append(55)`
  - `print(payoffs) -> [99, 0, 5, 1, 55]`
  - `payoffs.index(0) -> 1`
- Lists can be nested
  - `payoffs  = [[3, 0], [5, 1]]`
  - `payoffs[0] -> [3, 0]`
  - `payoffs[0][0] -> 3`

Looping Over Lists

- Constructing a for loop
  - `payoffs = [3, 0, 5, 1]`
  - `for e in payoffs:`

    `print(e*2) -> 3, 0, 5, 1`

- Using a list comprehension
  - `payoffs = [3, 0, 5, 1]`
  - `payoffs_double = [e*2 for e in payoffs]`
  - `print(payoffs_double) -> [6, 0, 10, 2]`
  - `payoffs_larger_one = [e for e in payoffs if e > 1]`
  - `print(payoffs_larger_one) -> [3, 5]`

Data Structures – Dictionaries

> Dictionaries are ordered in `key : value` pairs
  > `payoff_dict = {“cooperate” : 3, “defect” : 5}`
  > `payoff_dict[“cooperate”] -> 3`
  > `payoff_dict[“defect”] -> 5`

> More dictionary operations
  > `payoff_dict.keys() -> dict_keys(['cooperate', 'defect'])`
  > `payoff_dict.keys() -> dict_values([3, 5])`
  > `payoff_dict.items() -> dict_items([('cooperate', 3), ('defect', 3)])`
  > `payoff_dict[“nukes”] = 0`
  > `print(payoff_dict) -> {“cooperate” : 3, “defect” : 5, “nukes” : 0}`
  > `“cheat” in payoff_dict -> False`

> Note that keys must be unique and immutable

Looping Over Dictionaries

- Constructing a for loop
  - `payoff_dict = {"cooperate" : 3, "defect" : 5}`
  - `for k, v in payoff_dict.items():`
    `print(k, v) -> cooperate 3 defect 5`
- Using a dict comprehension
  - `payoff_dict = {"cooperate" : 3, "defect" : 5}`
  - `payoff_double_dict = {k : v*2 for (k, v) in payoff_dict.items()}`
  - `print(payoff_double_dict) = {"cooperate" : 6, "defect" : 10}`

Lists vs Dictionaries

- Lists
  - ordered sequence of elements
  - look up elements by an index integer
  - indicies have an order
  - index is an integer

- Dictionaries
  - matches keys to values
  - look up one item by another item
  - no order is guaranteed
  - key can be any immutable type

Decomposition and Abstraction

- Decomposition and abstractions are means to achieve good programming
- Decomposition
  - Concept: different devices work together to achiece an endgoal
  - Programming: break code up into self-contained modules that can be reused
  - Goal: ensure coherence, organization
  - Python: achieve decomposition through **functions** and **classes**
- Abstraction
  - Concept: do not need to know how device works to use it
  - Programming: code is black box, cannot see all details, do not want to see all details
  - Goal: provide adequate instructions for how to use code
  - Python: achieve abstraction with function **specifications** or **docstrings**

Functions

- Functions are resusable pieces of code
- Functions are not run in a program until they are “called” or “invoked”
- Characteristics of a function
  - name
  - parameters (0 or more)
  - docstring (optional but recommended)
  - body (expressions to be executed)
  - return value (the output of the function)

- Note that we have already used some of Python‘s built-in functions such as `len()`

Defining and Calling a Function

```
> def even_payoffs(x):
    """
    takes a list x as input and
    returns a list of all even values
    """
    even = [e for e in x if e%2 == 0]
    return even

> even_payoffs([3, 0, 5, 1) -> [0]
```

keyword

name

parameter

docstring

body (expression + return)

function call

Recursion

> Recursion is the process of repeating items in a self-similar way
  > Algortithmically this means: reduce a problem to simpler versions of the same problem
  > Semantically this mens: a programming technique where a function calls itself

> Example of a recursive function:
  >
```
def factorial(x):
    if x == 1:
        return 1
    else:
        return x*factorial(x-1)
```

Recursion – Execution

```
> def factorial(x):
      if x == 1:
          return 1
      else:
          return x*factorial(x-1)
```

base case

recursive step

```
> factorial(4) -> 24
```

4*(4-1)	3*(3-1)	2*(2-1)	1

return 4*6 ← return 3*2 ← return 2*1 ← return 1*1

Modules, Packages, Libraries

> A module is a .py file that contains functions that you intend to reuse
> A package is a directory of modules
> A library loosely refers to published packages

> In Python we use import <package_name> to access packages and modules

- import random → package
- random.randrange(0, 10)

- from random import randrange → module
- randrange(0, 10)

> Additional packages can be installed from the command line using pip install <package_name>

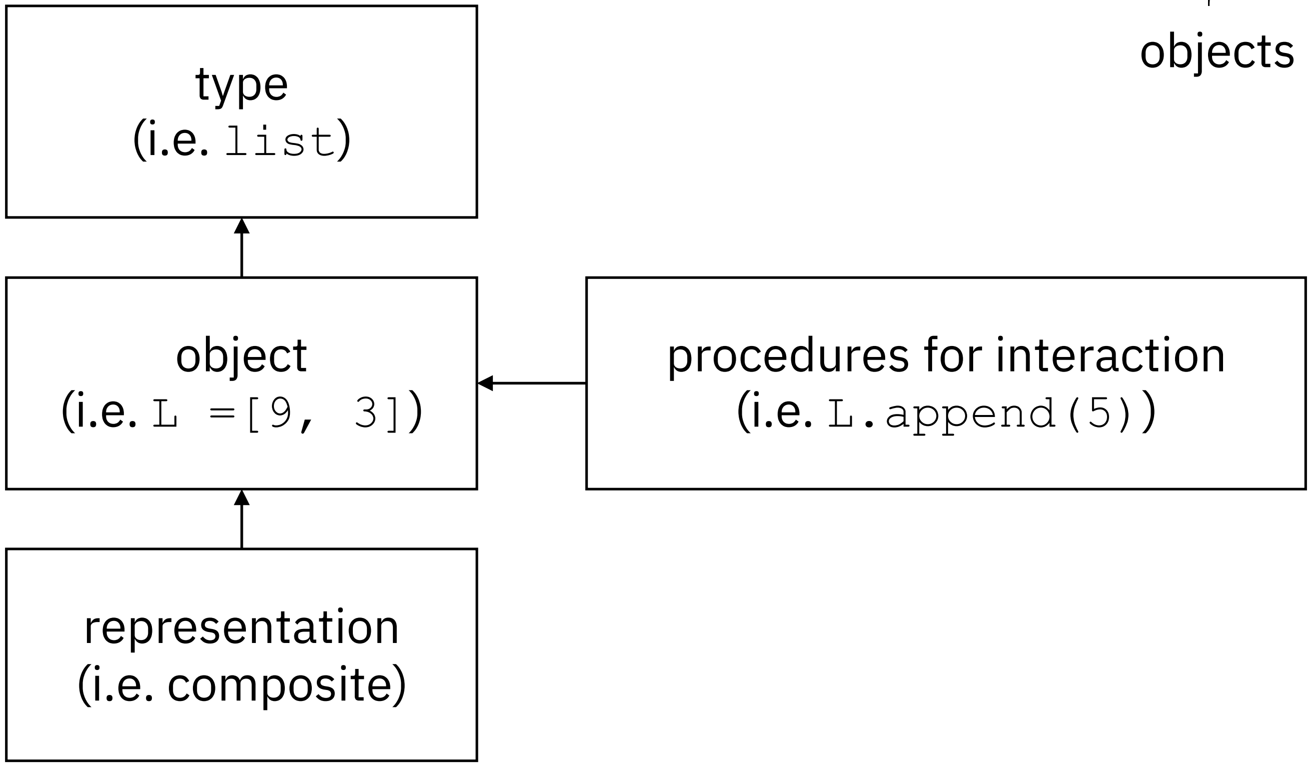
Understanding the Structure of a Program

> We have covered concepts and syntax that allow us to write simple programs
> A high number of rule-based StarCraft II agents are built around conditionals and iterations
> However, as these programs contain many blocks of conditionals and iterations, they are arranged according to a special paradigm that we need to understand
> This paradigm is called **Object Oriented Programming**
> In addition, the programs handel their tasks in an **asynchronous** manner
> We‘ll deal with each of these concepts in turn

Object Oriented Programming 1

> Central idea: everything in Python is an object (`2018, “Washington”, [9, 3]`)

objects



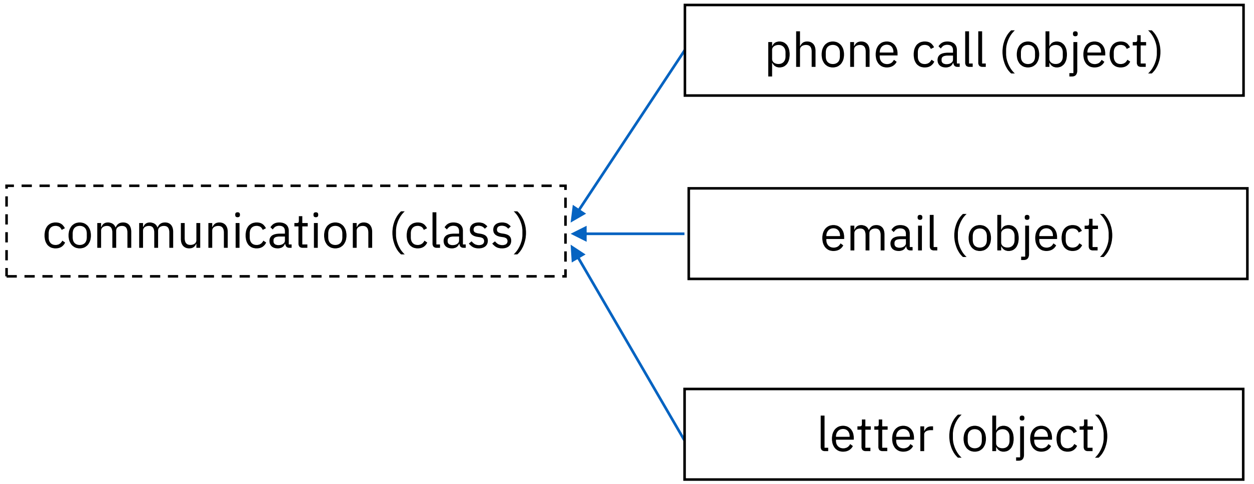
Object Oriented Programming 2

- Each object has
  - a **type**
  - an internal **data representation** (primitive or composite)
  - a set of procedures for **interaction** with the object

- An object is an instance of a type
  - `2018` is an instance of `int`
  - `L =[9, 3]` is an instance of `list`

Classes

> Every object is built from a class



> A **class** is a template or set of instructions to build a specific type of object

> An **instance** is an object build from a specific class

> A class can be a **subclass** of a **superclass**

Creating and Using Classes

- Distinguish between **creating a class** and **using an instance** of the class
- **Creating** the class involves
  - defining the class name
  - defining class attributes
- **Using** the class involves
  - creating new instances of objects
  - doing operations on the instances
  - i.e. `L = [9, 3]` and `L.append(5)`

Defining New Types

keyword

name / type

class parent

```
> class Communication(object):
    # define attributes here
```

- The word `interaction` means that `Communication` is a way of interaction and **inherits** all the attributes of `interaction`
  - `Communication` is a sublass of `interaction`
  - `interaction` is a superclass of `Communication`

Attributes

> Attributes are data and procedures that **belong** to the class
> **Data attributes**
>> think of data as other objects that make up the class
>> i.e. communication is made of phone call or email or letter and two or more humans
> **Methods** (procedural attributes)
>> think of methods as functions that only work with this class
>> how to interact with the object
>> i.e. you can define a rhetorical question between two humans over a phone call but not between two clouds in the sky

Defining How to Create an Instance of a Class

> We use a special method called `__init__` to initialize data attributes

special method to create an instance

```
> class Communication(object):
      def __init__(self, x, y):
          self.x = x
          self.y = y
```

data that initializes a `Communication` object

parameter to refer to an instance of an instance of the class

two data attributes for every `Coordinate` object

Creating an Instance of a Class

```
> c = Communication("Nitze", "Herter")
> print(c.x) -> "Nitze"
> private = Communication ("You", "Me")
> print(private.y) -> "Me"
```

create a new object of type Communication and pass in `"Nitze"` and `"Herter"` to the `__init__`

use the dot to access an attribute of instance `private`

Methods

- A method is a procedural function that only works with a specific class
- Python always passes the object as the first argument
  - Convention is to use `self` as as the name of the first argument of all methods

- The “`.`” operator is used to access any attribute (data or method)

Defining a Method for `Communication`

```
> class Communication(object):
      def __init__(self, x, y):
          self.x = x
          self.y = y
      def allies(self, other):
          allies1 = [self.x, other.x]
          allies2 = [self.y, other.y]
          return allies1, allies2
```

use `self` to refer to any instance

another parameter to method

dot notation to access data

Using the Method

```
> c = Communication("Nitze", "Herter")
> d = Communication("Mickey", "Donald")
> print(c.allies(d)) -> ["Nitze","Mickey"] ["Herter", "Donald"]
```

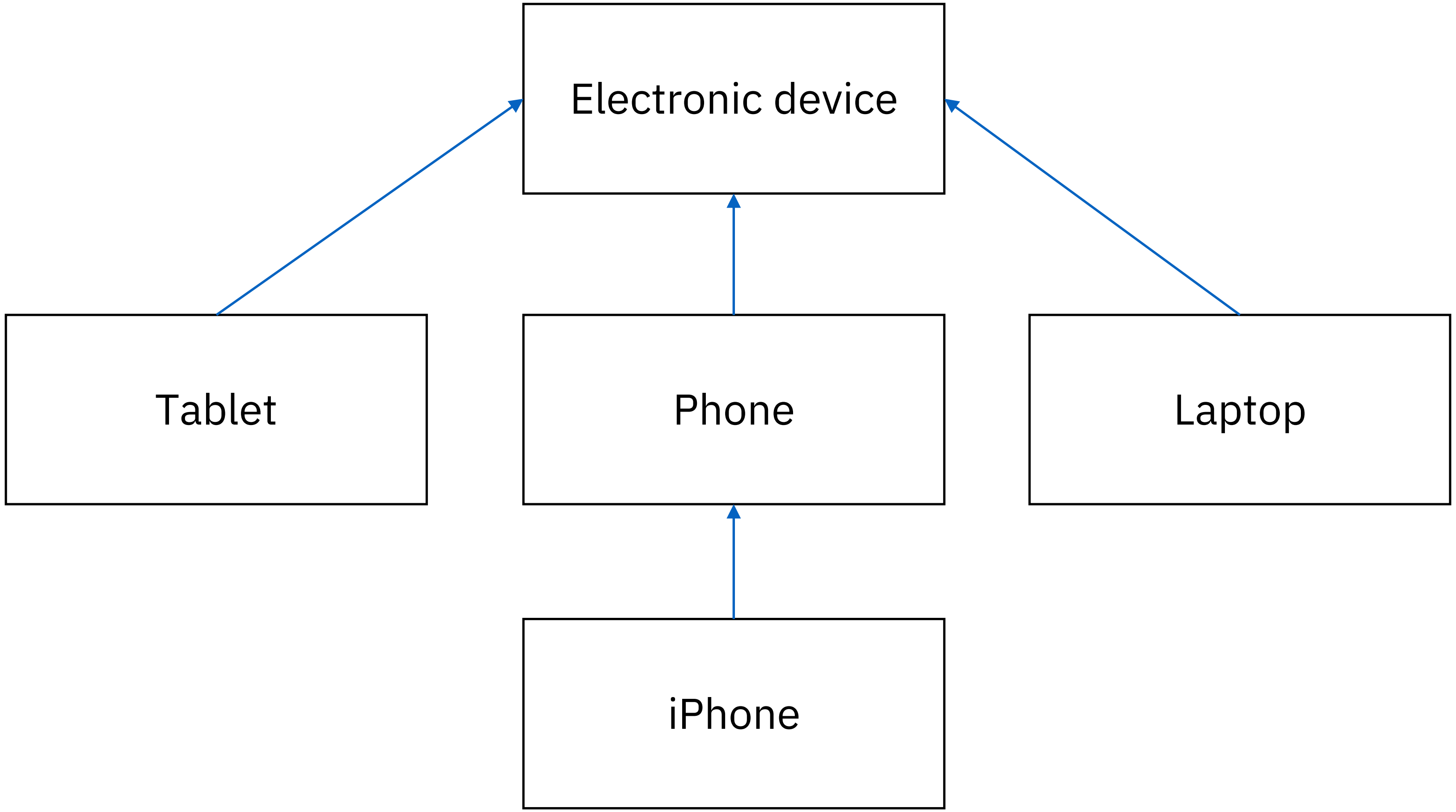
object to call method on

name of method

parameters (not including `self`)

Hierarchies and Inheritance

- **Parent class** (superclass)
- **Child class** (subclass)
  - Inherits all data and behavior from parent class
  - Add more info
  - Add more behvaior
  - Override behavior



Object Oriented Programming Recap

- Create your own **collections of data**
- **Organize** information
- **Division** of work
- Access information in a **consistent** manner
- Add **layers** of complexity
- Like functions, classes are a mechanism for **decomposition** and **abstraction**