



# **COMPUTATIONAL APPLICATIONS TO POLICY AND STRATEGY (CAPS)**

Session 2 – Python Primer

Leo Klenner

# Outline

1. Recap
2. Why Python?
3. Python Basics
4. Object Oriented Programming
5. Asynchronos Programming
6. Required Software



# Recap – Agent-Environment Interaction



Screenshot from AlphaGo's match against world champion Ke Jie

# Core Points

- > Agent-environment interaction framework for AI
- > Different degrees of interaction (based on explicit rules or adaptation)
- > Each model (expert system, reinforcement learning, ...) has its own constraints
- > These models provide powerful augmentations of human decision making
- > Advances in AI have consequences for the global order



# Recap Question

- > How would you characterize the sources of uncertainty in Go?
- > How does your answer affect what model you would chose to build a Go-agent?

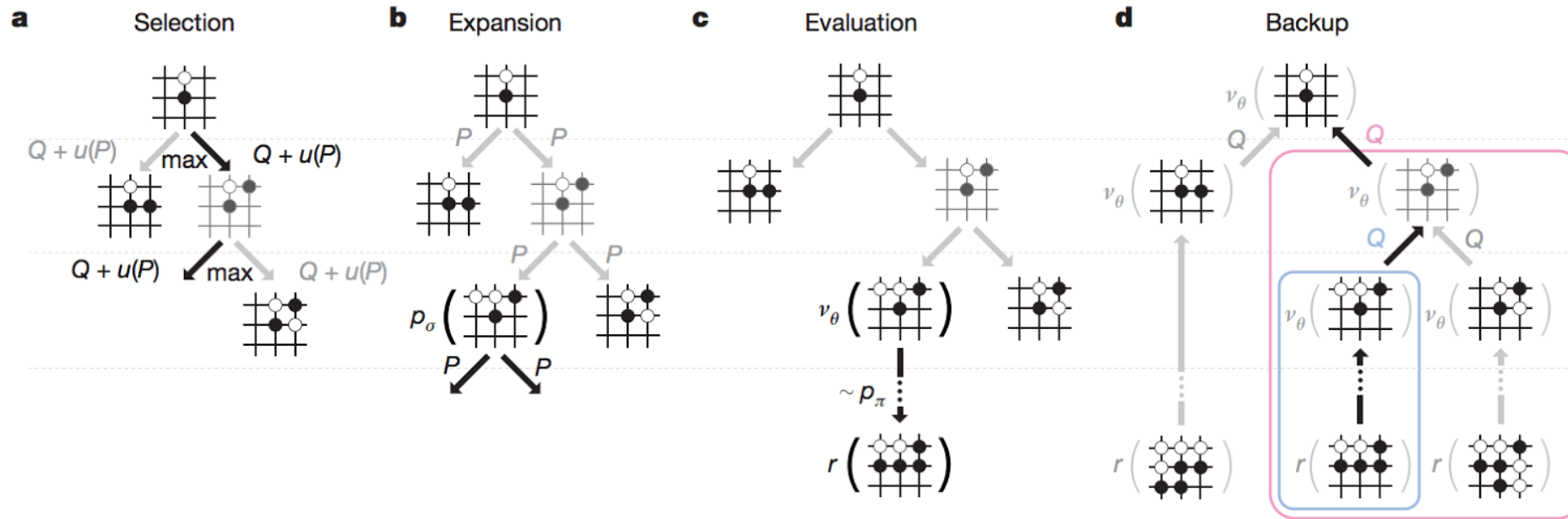


# Two Sources of Uncertainty in Go

- > Stern, Graspel, MacKay. 2008. Modelling Uncertainty in the Game of Go:
  - > Search-space uncertainty. Although Go is a game of perfect information (for each position, all possible moves are known), there are 200-300 possible moves per position
  - > Evaluation uncertainty. A Go stone has no intrinsic value; value is determined by relationship to other stones that is hard to formalize



# Uncertainty Reduction in AlphaGo



- > AlphaGo's backbone is a Monte Carlo Tree Search (MCTR) algorithm (figure)
- > MCTR looks ahead by simulating the remaining game play many times over
- > AlphaGo augments MCTR with two deep neural networks that evaluate positions and enable guided search to narrow down the uncertainty arising from Go's high branching factor



# Understanding Computation

- > To understand in more detail how agents interact with an environment we need some knowledge of programming
- > Programming is useful both to build solutions and equally as a way of thinking about problems even if your solution is qualitative





# Why Python?

- > Open source general-purpose language
- > Supports object oriented, procedural, functional programming styles
- > Interactive console
- > Highly readable syntax and Zen-like best practices (`import this`)
- > Additional resources:
  - > <https://github.com/Akuli/python-tutorial>
  - > <https://www.edx.org/course/introduction-to-computer-science-and-programming-using-python>



# 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:
```

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

condition

consequent (note the indentation)

```
> defect = 5
```

```
if defect == 3:
```

```
    print(defect) -> # defect > 3, consequent will not be executed
```





# Conditionals – else

> We need `else` to express the alternative of the condition in our `if` statement

> `if + else`

> `cooperate = 3`

> `if cooperate > 3:`

`print("I'll cooperate")`

`else:`

→ takes no condition

`print("I'll defect")` -> I'll defect



# 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
```

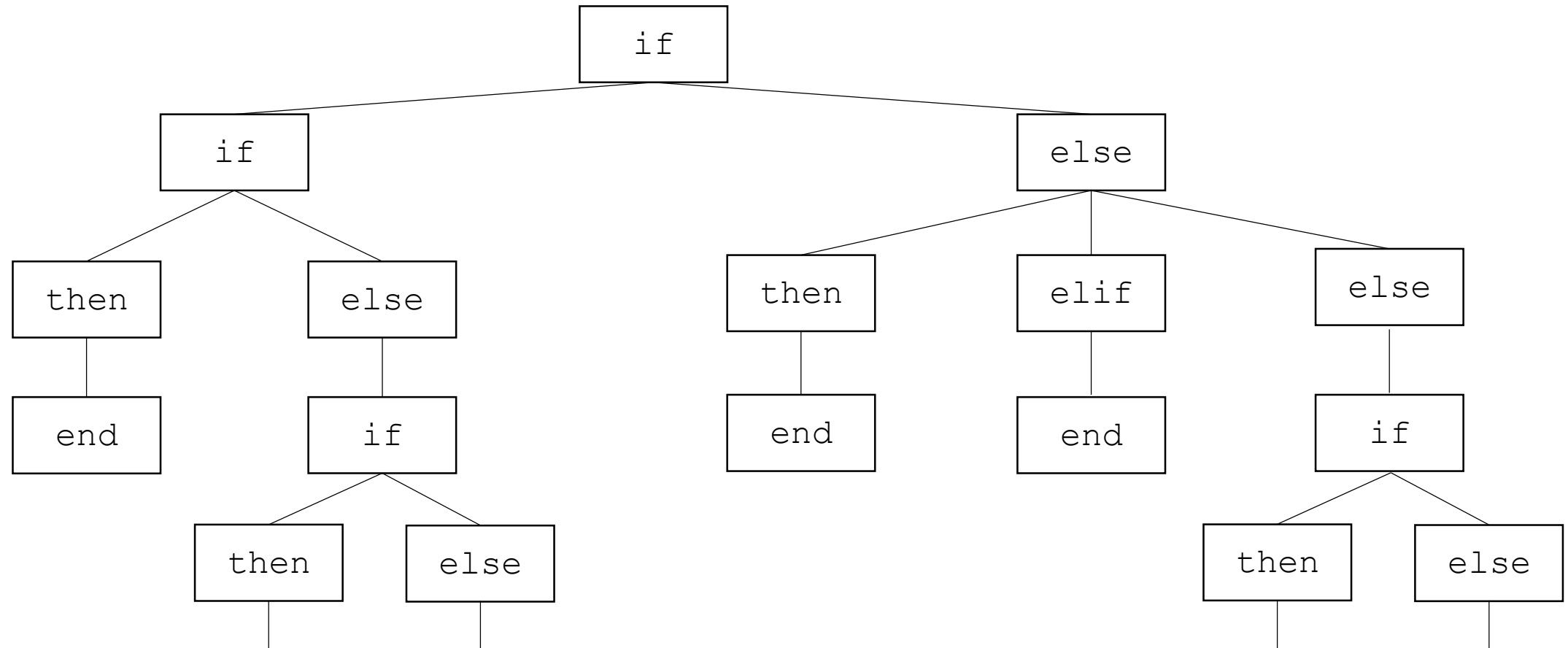
```
else: → alternative of A and B
```

```
    print("I'll defect")
```



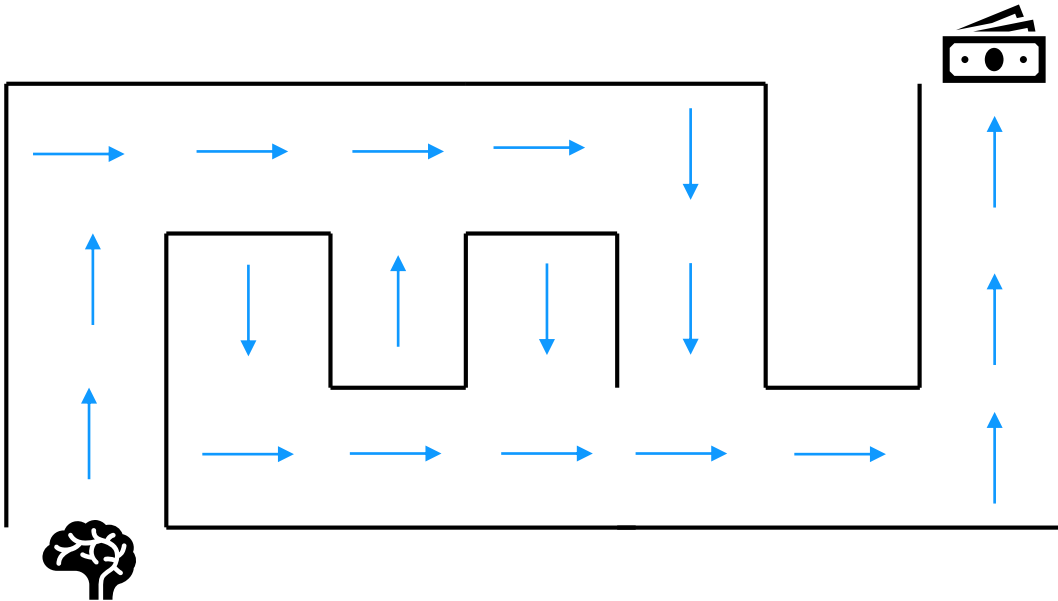
# Working with Conditionals

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



# Solving a Simple Pathfinding Problem 1

- > We can use our conditional logic to solve a simple pathfinding problem



- > If right clear, go right
- > If right blocked, go forward
- > If right and front blocked, go left
- > If right, front, left blocked, go back
- > If position == reward, end



# Solving a Simple Pathfinding Problem 2

- > Conditionals enable our agent to evaluate her position and move to the reward
- > But to completely traverse the path we need to nest a lot of conditionals
- > Iteration provides an efficient way around this problem



# 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 inside 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 smallest value
- > next time, `<variable>` gets the prev value +1
- > etc.



# Applying 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 not 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.values() -> 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
- > indices 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 achieve 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**



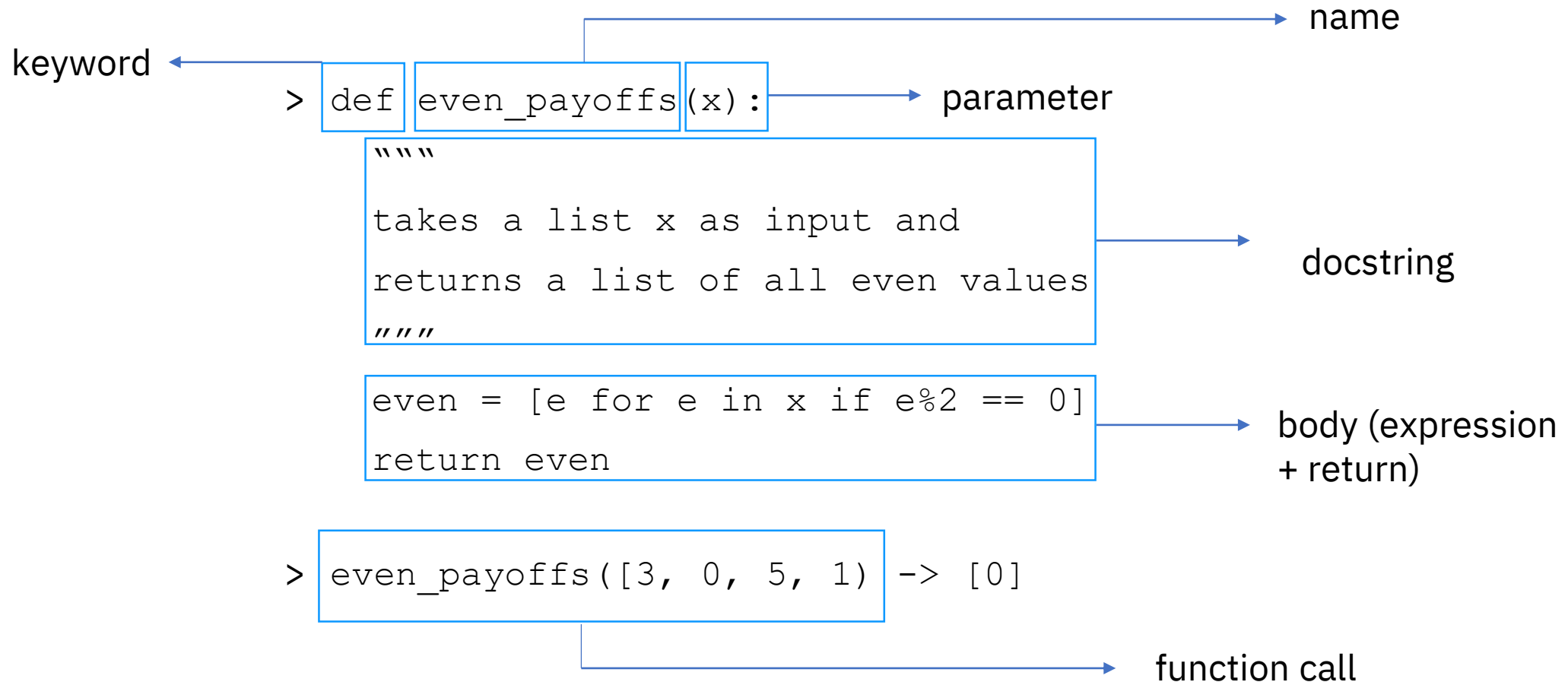
# Functions

- > Functions are reusable 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



# Recursion

- > Recursion is the process of repeating items in a self-similar way
  - > Algorithmically this means: reduce a problem to simpler versions of the same problem
  - > Semantically this means: 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
```

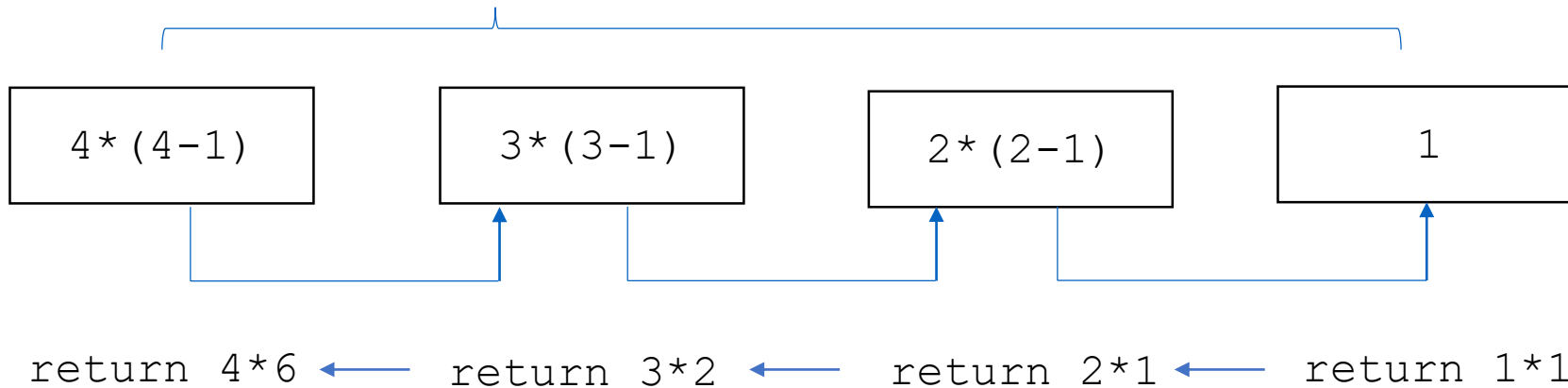
base case

```
    else:
```

```
        return x*factorial(x-1)
```

recursive step

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



# 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>`



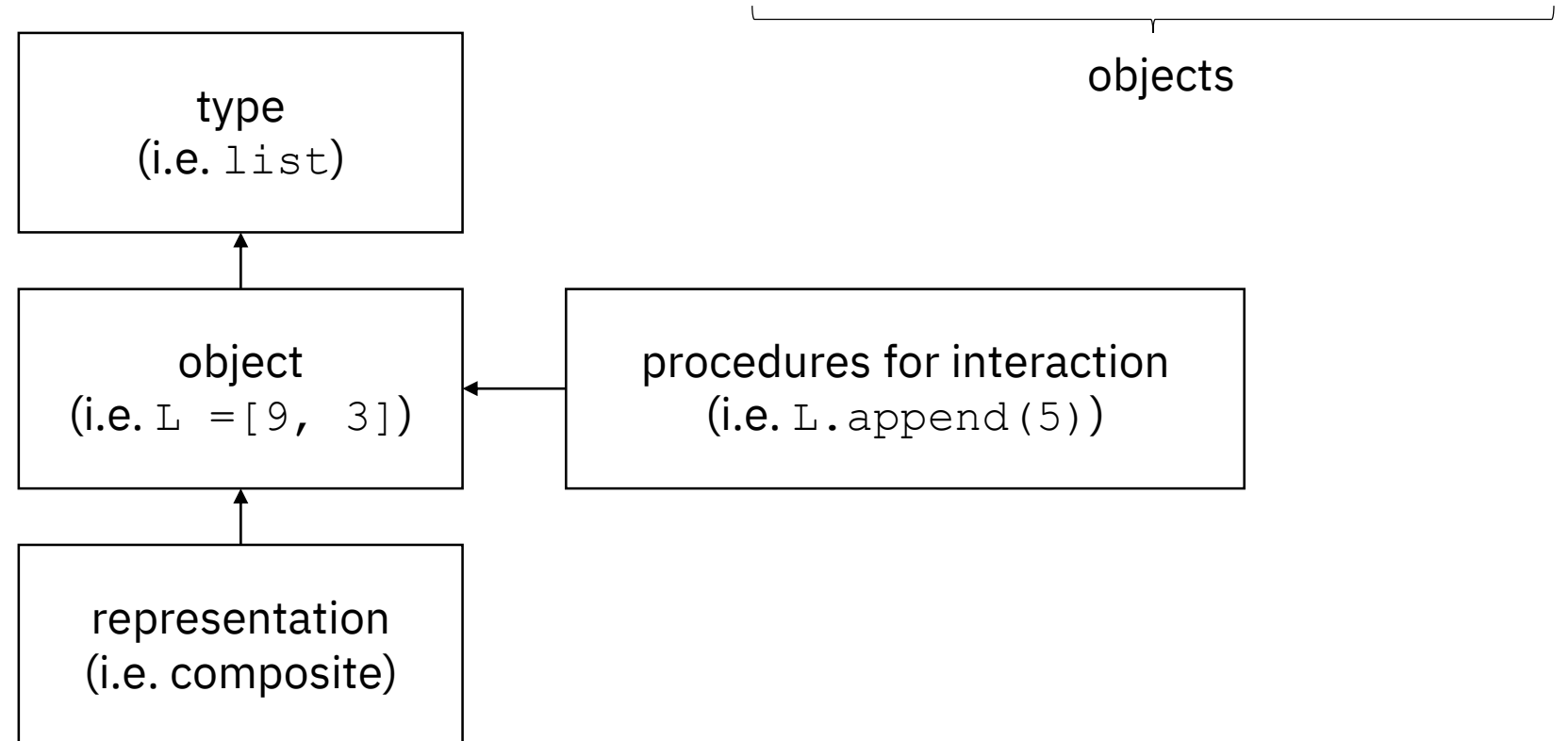
# Understanding the Structure of a Program

- > 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 handle 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]`)



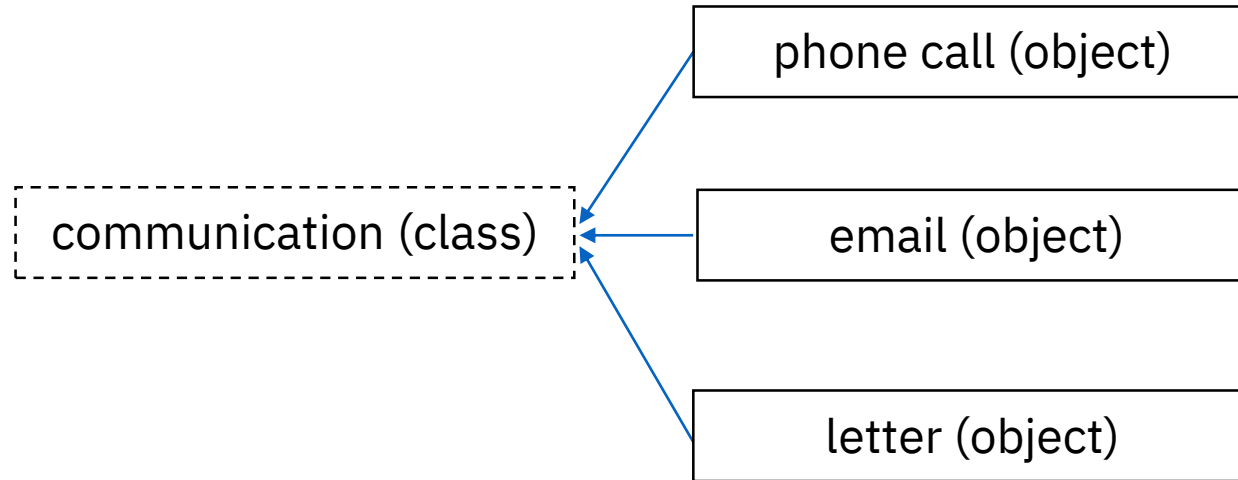
# Object Oriented Programming 2

- > 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 ← > class Communication(interaction) : name / type  
class parent  
# define attributes here

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



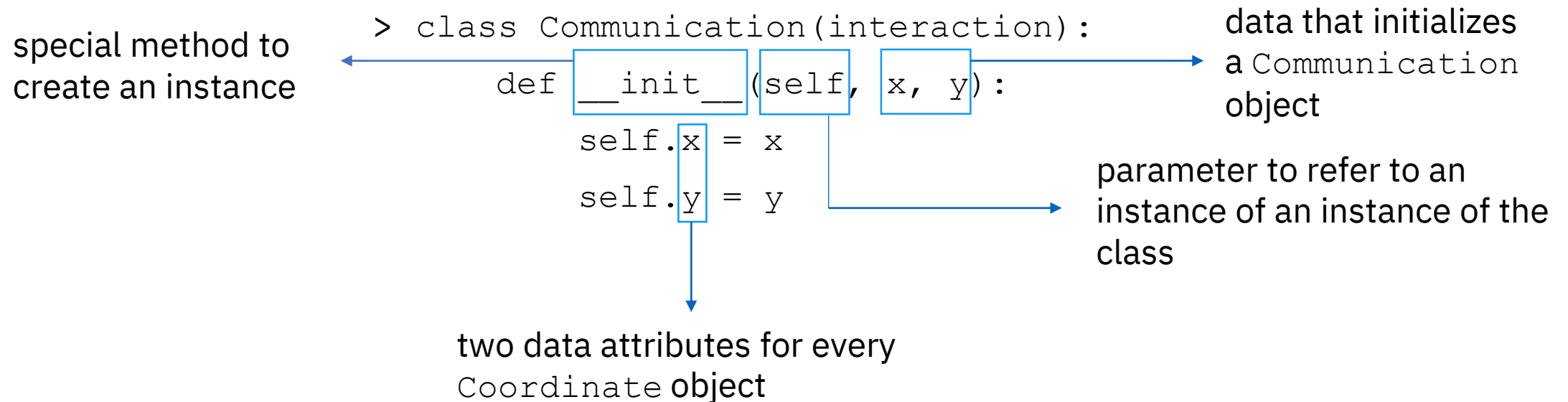
# Attributes

- > 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



# 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 (in our example, `interaction`)
  - > Convention is to use `self` 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(interaction):  
    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`)



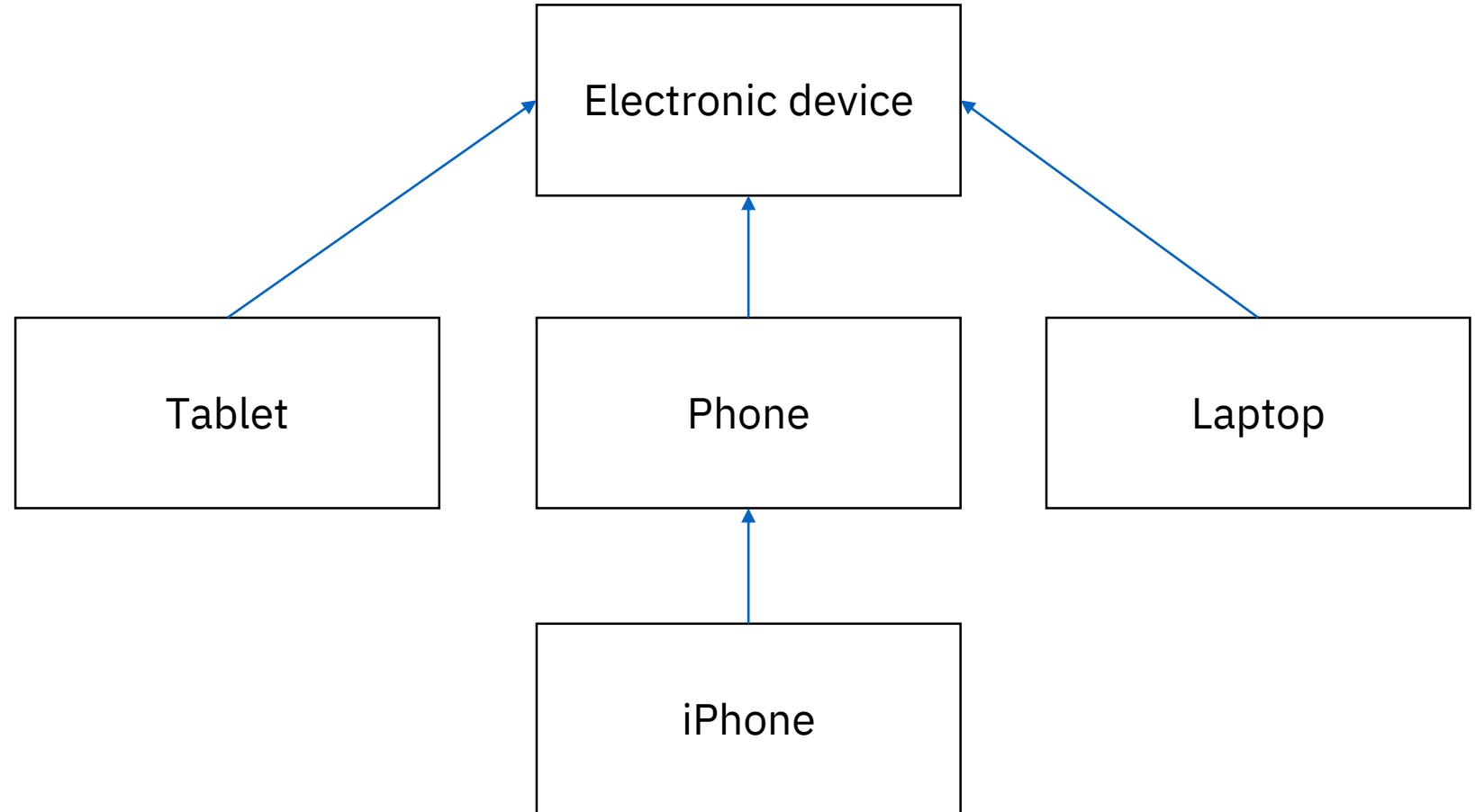


# Hierarchies and Inheritance

> **Parent class**  
(superclass)

> **Child class**  
(subclass)

- > Inherits all data and behavior from parent class
- > Add more info
- > Add more behavior
- > 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**



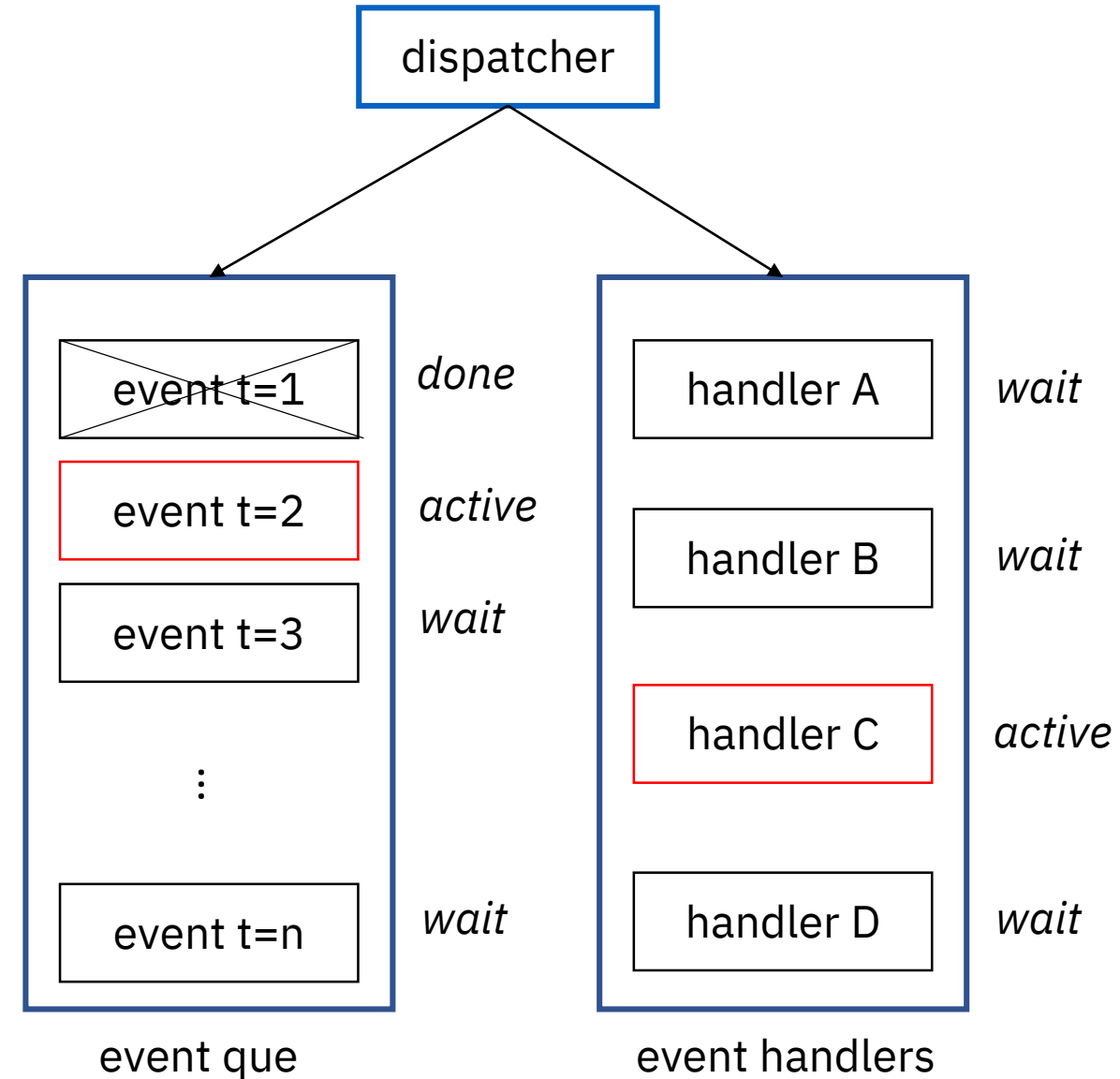
# An Assumption About Program Execution

- > Program is executed line by line, one line at a time
- > Each time a function is called, program execution waits until that function returns before continuing to the next line of code
- > Referred to as **synchronous** programming
- > Trade-offs
  - > if a function is called and starts a time consuming process, the program has to wait until the process is complete
  - > “wait at every step” even if steps are independent can diminish the program’s responsiveness and performance



# Synchronous Flow

- > Events (data) are queued up according to time  $t$  of their occurrence
- > Events are processed by event handlers (blocks of code)
- > **Dispatcher** assigns control of program to either que or handlers
  - > record event (**que**)
  - > process event (**handler**)
- > At time  $t$  all handlers (except one) and all events from  $t=t + 1...n$  are waiting for event  $t$  to finish processing



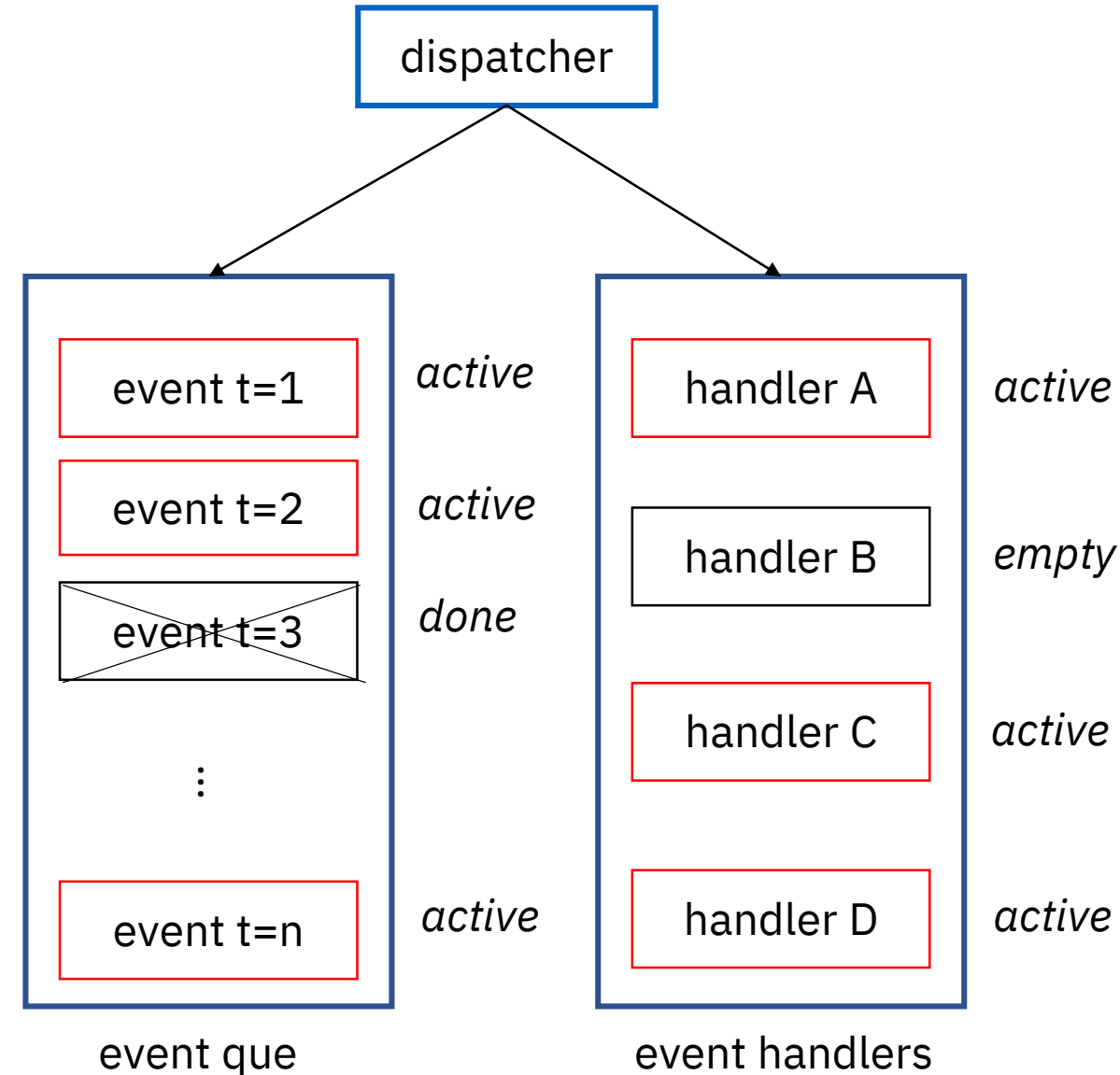
# Asynchronous Programming

- > In **asynchronous** programming we switch from “waiting at every step” to “waiting at the end for all steps to complete”
  - > If a function is called and starts a time consuming process, we “spend our waiting time” by moving to the next step and again repeat this behaviour until the last step is reached
- > We wait at the last step until all preceding steps have been executed



# Asynchronous Flow

- > At time  $t=n$  all events that have occurred up to  $n$  are being processed (exception: waiting for identical handler)
- > This is not done in parallel but in **rapid alternation between functions**
- > The order of execution is **no longer predictable** (event  $t=3$  executed before event  $t=1$ )
- > Instead, the order of execution depends on the events generated



# Summary

- > Python allows us to perform operations on **primitive data types** such as strings and numbers (integers, floats)
- > Primitive types can be aggregated into **composite data types** (lists, dictionaries) that differently access, management and manipulate data
- > **Conditionals** (if, elif, else) allows us to structure our program around conditions that need to be satisfied for code to be executed
- > **Iteration** (for, while) and **recursion** provide further mechanisms for structuring our program depending on the task we need to achieve
- > **Functions** and **classes** allow the abstraction and decomposition of our code
- > **OOP** allows division of work and layered complexity in our program
- > **Asynchronous programming** allows dealing with a high number of events



# Required Software

- > For next week you only need StarCraft II and python-sc2
- > Python-sc2
  - > via command line: `pip install sc2`
- > StarCraft II (free to play)
  - > <https://us.battle.net/account/download/>
  - > This will take some time so best to do it at home
- > StarCraft II maps
  - > <https://github.com/Blizzard/s2client-proto#map-packs>
  - > Once installed, extract the maps as subdirectories into StarCraft II's map directory
  - > Do this after you have installed StarCraft II

