COMPUTATIONAL APPLICATIONS TO POLICY AND STRATEGY (CAPS)

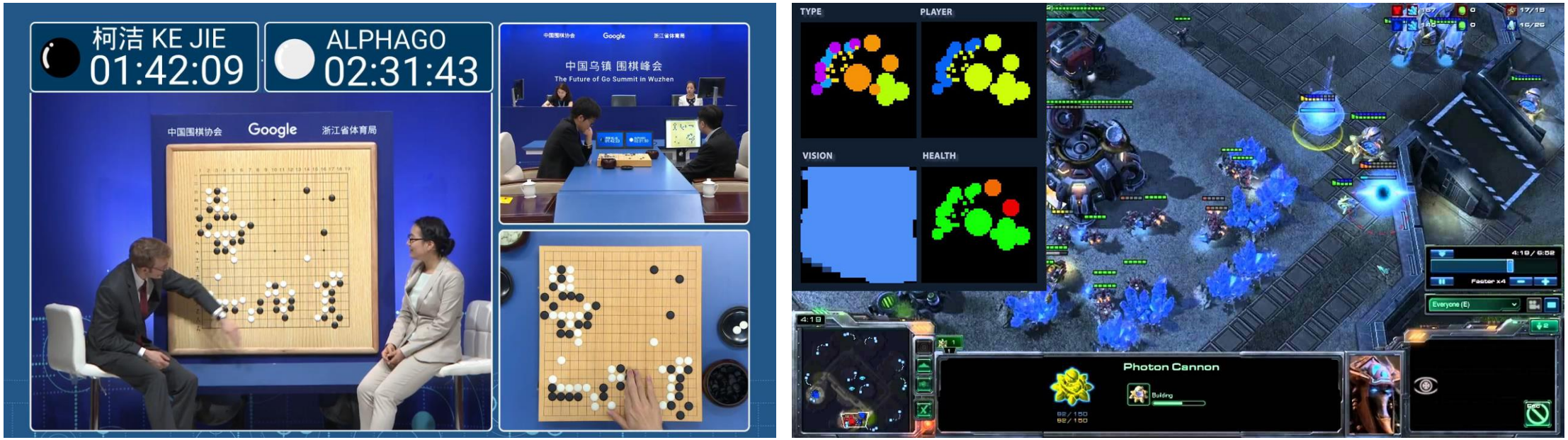
Session 2 – Python Primer

Leo Klenner

Outline

1. Recap Session 1
2. Short Note on Programming
3. Learning Python Through Reverse Engineering a Bot
4. WorkerRushBot – Block 1
5. WorkerRushBot – Block 2
6. WorkerRushBot – Block 3
7. Required Software

Recap – AI Agent-Environment Interaction



Screenshots from AlphaGo‘s match against world champion Ke Jie and DeepMind‘s pysc2 API for StarCraft II

Core Points

> AI as model of agent-environment interaction based on rational agency
> Different degrees of interaction (based on explicit rules or learning / 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

Combining Rule-Based and Learning Agency

- Hybrid approaches are possible and can be highly useful to focus learning
- Hybrid approach implemented in Tencent‘s StarCraft bot
- Peng Sun et al. 2018. TStarBots: Defeating the Cheating Level Builtin AI in StarCraft II in the Full Game

How?
- Rule-based aspect 1: hard-coded dependency rules of SCII (i.e. unit **z** requires building **y**, building **y** requires building **x**) are encoded in the learning algorithm
- Rule-based aspect 2: high number of decisions in SCII are trivial (i.e. which worker of workers **w** builds **x**) and unnecessarily reduce learning speed, hence hard-coded embedding

What?
- Learning-based aspects: strategic dimensions such as what to build, when to attack etc.

Setting the Expectations

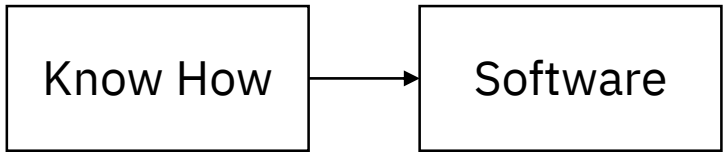
- Learning programming takes time and practice
- Steep initial learning curve
- Don‘t get discouraged
  - Think beyond code, i.e. integrate computational understanding into critical analyses
- Focus on navigating and dealing with unknowns
- Don‘t focus on trying to know everything

Goals

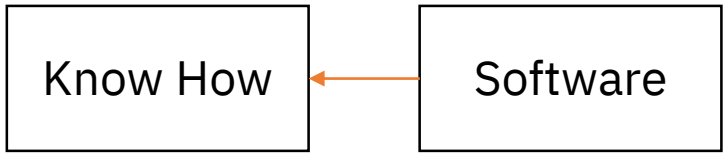
- Goal 1: understand 12 lines of code
  - Understand the **syntax** and the **concepts** the syntax is grounded in
- Goal 2: develop foundation to scale from 12 lines to 100+ next week
  - Understand which elements of the code you can ignore

Approach

> Engineering vs. reverse engineering



Engineering



Reverse engineering

> This session: decompose code to build up knowledge

> Next session: apply this knowledge to write code

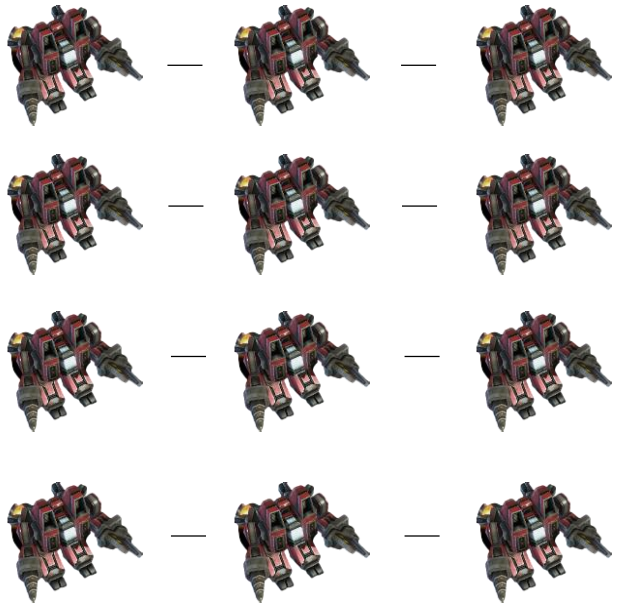
Reverse Engineering

WorkerRushBot.py

next week

Components → Concepts → Exercise

Worker Rush Strategy



> Endowed with 12 workers at t=0, send all workers to attack enemy base

Worker Rush Code

```
import sc2
from sc2 import run_game, maps, Race, Difficulty
from sc2.player import Bot, Computer

class WorkerRushBot(sc2.BotAI):
    async def on_step(self, iteration):
        if iteration == 0:
            for worker in self.workers:
                await self.do(worker.attack(self.enemy_start_locations[0]))

run_game(maps.get("Abyssal Reef LE"), [
    Bot(Race.Terran, WorkerRushBot()),
    Computer(Race.Protoss, Difficulty.Medium)
], realtime=True)
```

Developing Intuition

```python
import sc2
from sc2 import run_game, maps, Race, Difficulty
from sc2.player import Bot, Computer
```

1

```python
class WorkerRushBot(sc2.BotAI):
    async def on_step(self, iteration):
        if iteration == 0:
            for worker in self.workers:
                await self.do(worker.attack(self.enemy_start_locations[0]))
```

2

```python
run_game(maps.get("Abyssal Reef LE"), [
    Bot(Race.Terran, WorkerRushBot()),
    Computer(Race.Protoss, Difficulty.Medium)
], realtime=True)
```

3

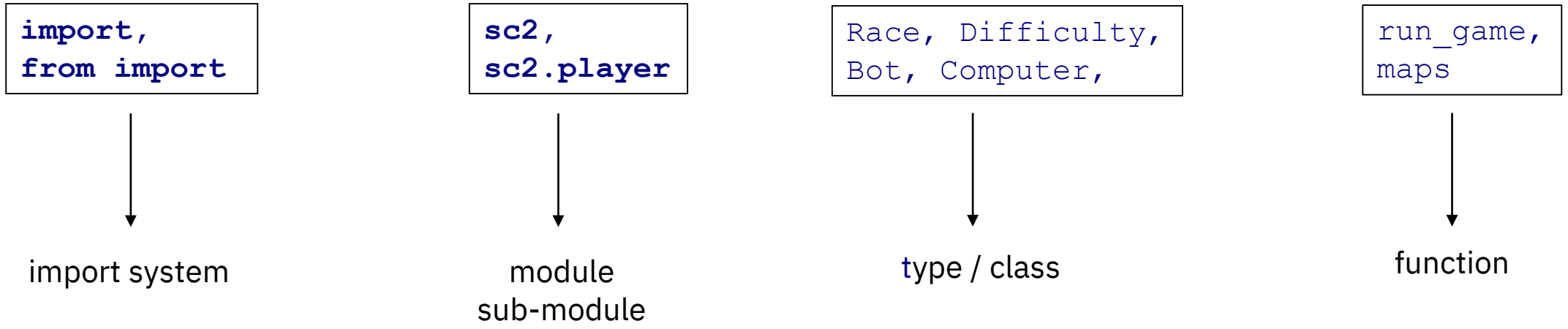
> Three blocks of code, each with different structure => seperate purpose

Tools

- Python has some built-in tools to help us in our analysis
  - `type(<object name>)` → what kind of (type of) object are we dealing with?
  - `dir(<object name>)` → what‘s inside objects of a specific type?

Block 1 – Analysis

```
import sc2
from sc2 import run_game, maps, Race, Difficulty
from sc2.player import Bot, Computer
```



- Intuition – selecting our sources
  - To write our bot, we first import code written by others into our program. This code is decomposed into different abstracted units (modules, classes, functions)

Modules

- Organizational unit of Python code containing various objects (submodules, classes, fuctions)
- Accessed through `import` statement, `import from` moves into submodule (i.e. directly imports a class or function)
- How to see the contents of a module? Use `dir(<module name>)`
- Access content of module use `<module>.<object>` i.e. `sc2.player`

Overview of Functions and Classes

> **Functions** are blocks of code that can be used repetitively in a program (instead of writing the block of code every time, simply call the function)

> **Classes** are collections of data and functionality (for our purposes, think of them as an aggregate of multiple functions)

> We can **use** pre-written functions and classes (i.e. `sc2.player.Bot`) or we can **define** our own functions and classes

Abstraction

> Abstration – do not need to or want to know how an object is working to use it
> Example – what is inside the function `sc2.run_game`?

Inside `sc2.run_game`

```
def run_game(map_settings, players, **kwargs):
    if sum(isinstance(p, (Human, Bot)) for p in players) > 1:
        join_kwargs = {k: v for k,v in kwargs.items() if k != "save_replay_as"}

        portconfig = Portconfig()
        result = asyncio.get_event_loop().run_until_complete(asyncio.gather(
            _host_game(map_settings, players, **kwargs, portconfig=portconfig),
            _join_game(players, **join_kwargs, portconfig=portconfig)
        ))
    else:
        result = asyncio.get_event_loop().run_until_complete(
            _host_game(map_settings, players, **kwargs)
        )
    return result
```

> Is this information valuable to us? It depends

Block 1 – Summary

```
import sc2
from sc2 import run_game, maps, Race, Difficulty
from sc2.player import Bot, Computer
```

> Connect our program to a set of sources (all originating from the `sc2` module) that allow it to gain the functionality of playing StarCraft 2

> Sources are organized in different units (submodule, classes, functions) that we can use without knowing their inside

> Functions and classes organize code and make it reusable

Exercise 1 – Using Modules and Functions

```
import random

dir(random)

rand_num = random.randrange(0, 100)

print(rand_num)

type(random)
type(random.randrange)
type(rand_num)
```

Block 2 – Intuition

```
class WorkerRushBot(sc2.BotAI):
    async def on_step(self, iteration):
        if iteration == 0:
            for worker in self.workers:
                await self.do(worker.attack(self.enemy_start_locations[0]))
```

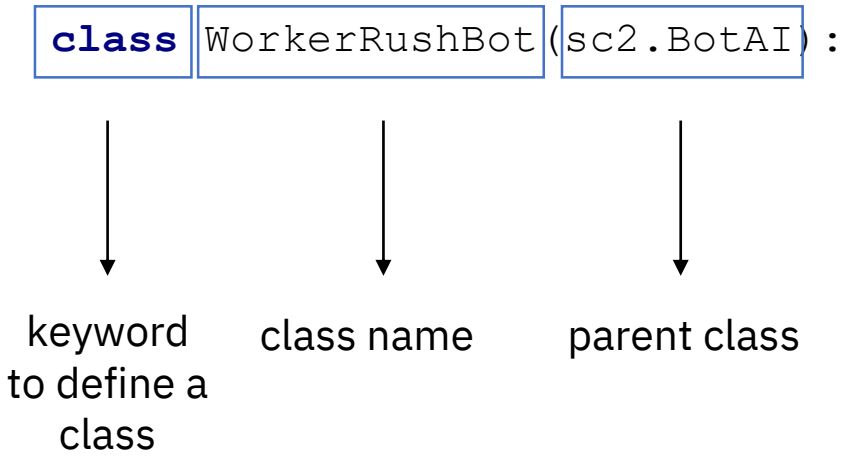
> More involved than the code in block 1 – what do we recognize?

Block 2 – Sub-Division

```
class WorkerRushBot(sc2.BotAI):
    async def on_step(self, iteration):
        if iteration == 0:
            for worker in self.workers:
                await self.do(worker.attack(self.enemy_start_locations[0]))
```

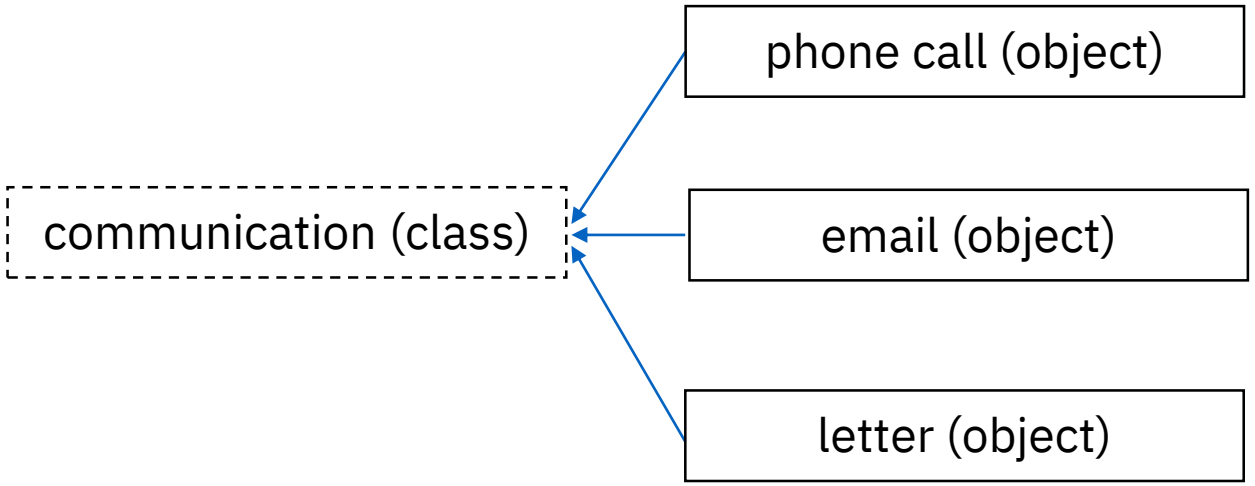
> Different levels of indentation – lower line dependent on upper line

Block 1 – Line 1



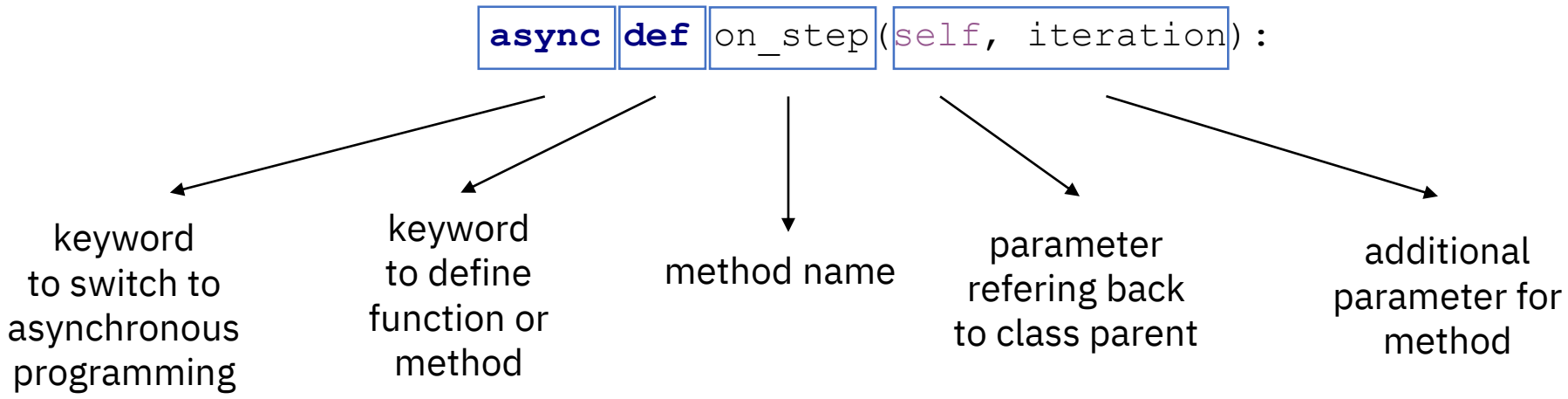
- Intuition – setting up our bot
  - Our bot (`WorkerRushBot`, later `CAPSbot`) is classified as an example of a `BotAI` in `sc2` and behaves according to rules specified for `BotAI`

Classes



- A **class** is a template or set of instructions to build a specific type of **object**
- A class can be a **subclass** (child class) of a **superclass** (parent class)
- The child class **inherits** all the attributes of the parent class
- Classes are used to structure our code efficiently and make it reusable

Block 2 – Line 2



- Intuition – ensuring that we move in lock-step with the game
  - We make our bot operational by providing it with a method to run on every game step of StarCraft

Methods

- A **method** is a function that works only within a specific class (hence, we use the `self` parameter to refer back to the parent class, here `sc2.BotAI`)

- Use `def` to define a method (or function)
- Specify a name for the method
- Pass parameters that we want the method to have access to
  - `self` is necessary and always the first parameter

- Add a body and return statement => code that is executed and returned when the method is called

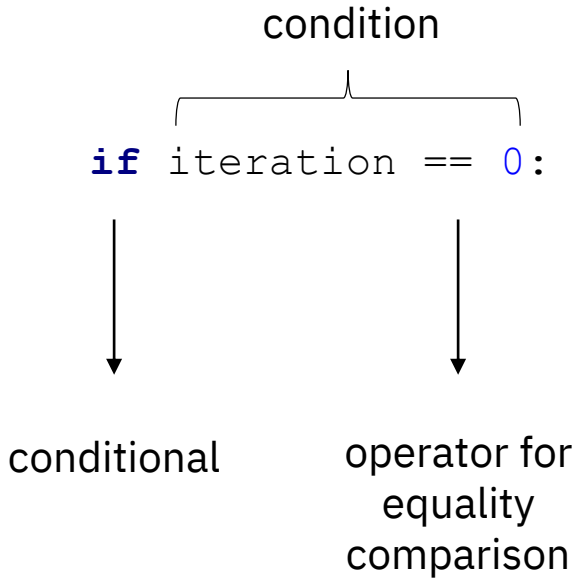
Asynchronous Programming

- We use **asynchronous programming** when our program has to proccess high volumes of irregular input data (StarCraft)
- Enables higher repsonsiveness to input data and hence higher performance

- How it works
  - **Synchronous** – assign one data point to method to be processed, wait until done, move to next data point etc.
  - **Asynchronous** – assign one data point to method to be processed, proceed to next data point etc., wait until all are done

- In Python, define a method as asynchronous using `async`
- Asynchronous methods (coroutines) are not called but **awaited** using `await`

Block 2 – Line 3



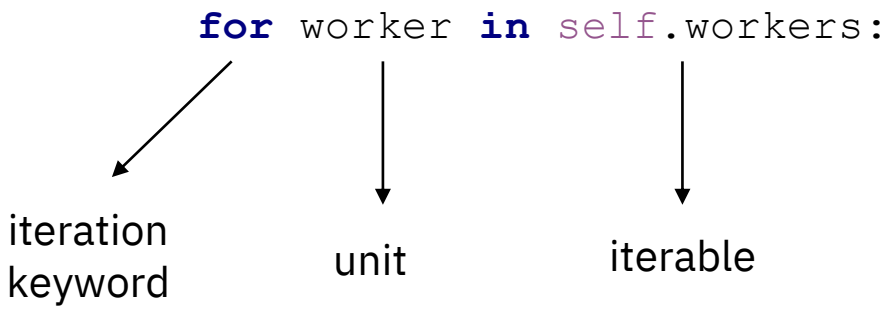
- Intuition – getting ready at time t = 0
  - We ensure that all following operations performed by our bot happen right at the start of the game when `iteration` (time measured in game steps) is equal to 0

Conditional Logic

- We use **conditional logic** to branch out operations in our program based on specified conditions and their alternatives
- Conditions evaluate to `True` or `False` and can be checked through comparative operators (`==`, `!=`, `>`, `<`, `>=`, `<=`)
- Python provide three conditional statements `if`, `elif`, and `else`

```
if iteration == 0:
    print("It is zero")
elif iteration == 1:
    print("It is one")
else:
    print("It is neither zero nor one")
```

Block 2 – Line 4



- Intuition – talking to all of our workers, one at a time
  - To send our 12 workers to the enemy base we have to select them first. Here we do this by selecting not 12 workers at once but selecting each worker in the total 12 workers at a time

Iteration with `for` loops

- Execute set of instructions n times where n > 1
- End when condition is reached
    - `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.

Block 2 – Line 5

```
await self.do(worker.attack(self.enemy_start_locations[0]))
```

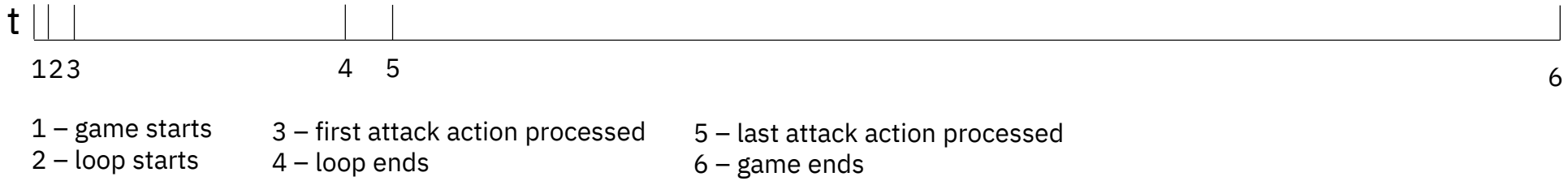
- Intuition – singaling the workers to attack
  - Each worker selected is sent to attack the first enemy start location

Flow of Processes in `WorkerRushBot`

game steps tracked through `on_step` (`async`)

if `on_step` parameter `iteration` is 0: start looping through workers

everytime a worker has been selected, interrupt `async` with `await` to process attack action



Lists

> Lists are one of Python‘s data structures
> Data structures are collections of values with specified relationships among them
> Lists are **ordered sequences** of elements and are **mutable**

```
SAIS_start_location = ["Bob", "Nitze", "Rome"]

SAIS_start_location[0] -> "Bob"
```

Exercise 2 – Conditionals, Iteration and Lists

```
hidden_coordinates = [55, 9, 3, 0, 101]
destination_coordinates = []

for coordinate in hidden_coordinates:
    if coordinate > 50:
        destination_coordinates.append(coordinate)
    else:
        pass

print("Our destination is at", destination_coordinates)
```

Block 3 – Analysis

```
run_game(maps.get("Abyssal Reef LE"), [
    Bot(Race.Terran, WorkerRushBot()),
    Computer(Race.Protoss, Difficulty.Medium)
], realtime=True)
```

> Exercise – we have the knowledge to read and understand this code

Reassembling Our Code

```
import sc2
from sc2 import run_game, maps, Race, Difficulty
from sc2.player import Bot, Computer

class WorkerRushBot(sc2.BotAI):
    async def on_step(self, iteration):
        if iteration == 0:
            for worker in self.workers:
                await self.do(worker.attack(self.enemy_start_locations[0]))

run_game(maps.get("Abyssal Reef LE"), [
    Bot(Race.Terran, WorkerRushBot()),
    Computer(Race.Protoss, Difficulty.Medium)
], realtime=True)
```

load source components

create bot and specify operations

initialize game environment and execute bot

Summary

> We decomposed a bot desinged to play StarCraft 2
> Core components we identified are: modules, classes, methods, functions, operators, conditionals, iteration, lists and asynchronous programming
> We analyzed the pattern of process execution in `WorkerRushBot`
> We have acquired the knowledge to write our own bot next week
> Keep in mind – programming is hard, navigate the unknowns, relax, have fun

Required Software

- For next week you 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

Keep Learning

> Additional, more comprehensive slides on Python on the course website