MCPI

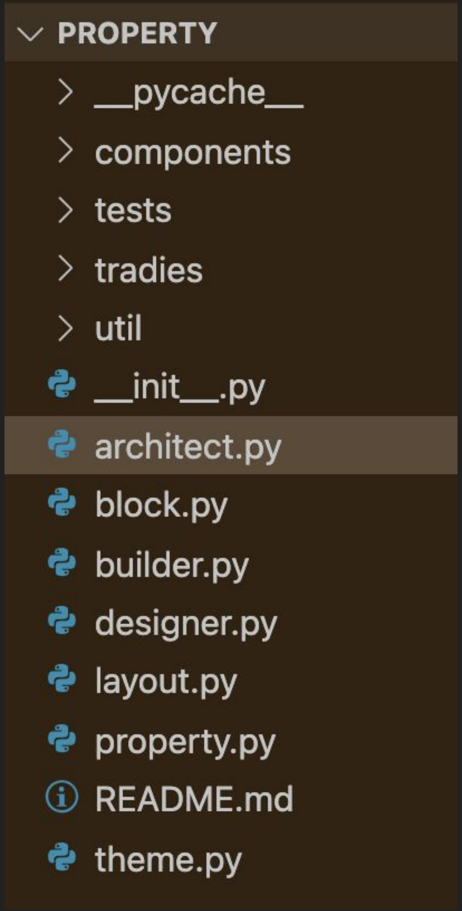
COSC2804 Assignment 1

Introduction - Lachie

House Generation

Jin

- Architect class:
    - Orchestrates property generation



PROPERTY

- __pycache__
- components
- tests
- tradies
- util
- __init__.py
- architect.py
- block.py
- builder.py
- designer.py
- layout.py
- property.py
- README.md
- theme.py

House Generation

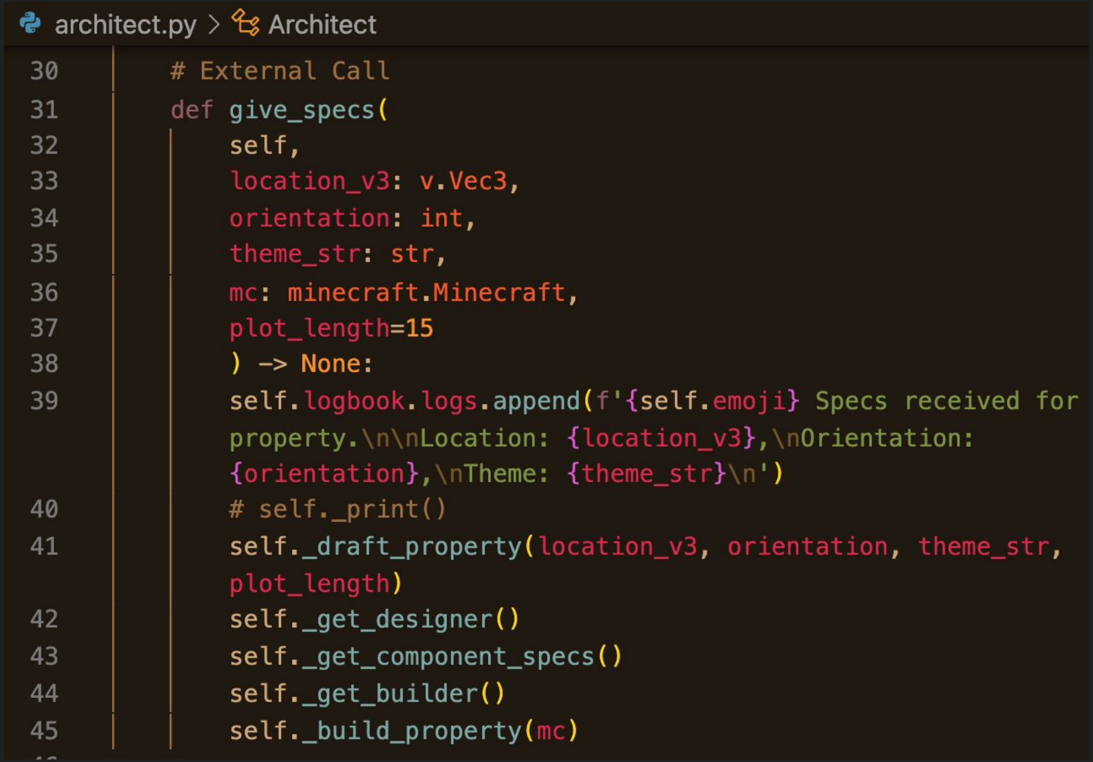
Jin

- Architect class:
    - Receives property details via external call
        - Vec3, orientation, theme
    - Instantiates a Property object
    - Creates a Layout for the Property
    - Calls upon Designer to instantiate various components for the property
    - Calls upon Builder to build out those components

House Generation

Jin

- Architect class:
    - External Call



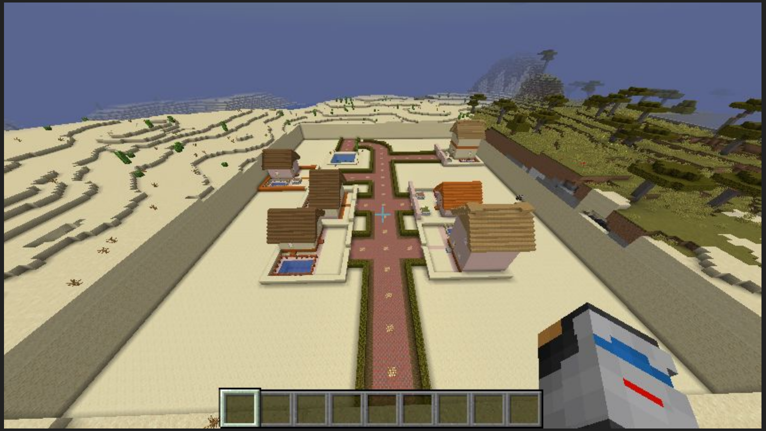
```python
    # External Call
    def give_specs(
        self,
        location_v3: v.Vec3,
        orientation: int,
        theme_str: str,
        mc: minecraft.Minecraft,
        plot_length=15
        ) -> None:
        self.logbook.logs.append(f'{self.emoji} Specs received for property.\n\nLocation: {location_v3},\nOrientation: {orientation},\nTheme: {theme_str}\n')
        # self._print()
        self._draft_property(location_v3, orientation, theme_str, plot_length)
        self._get_designer()
        self._get_component_specs()
        self._get_builder()
        self._build_property(mc)
```

Village Foundation

Lachie

- Buildings in the village all exhibit working foundations
- None of the builds present in the village have terrain or unexpected blocks seeping into them.
- Due to this, the buildings blend harmoniously into the game world

The buildings in the village are supported by foundations and blend harmoniously into the game world

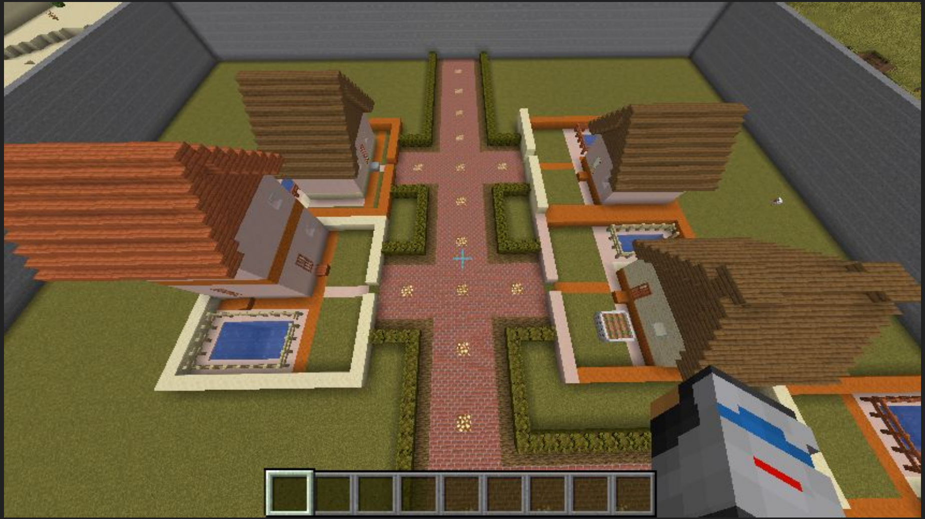


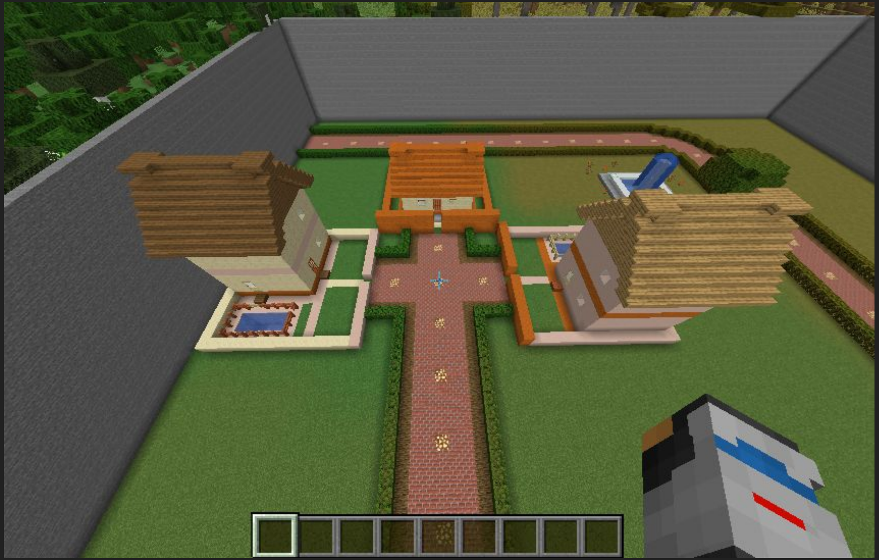
Roads

Lachie

- Roads are always connected to other roads and to other houses (roads are never off by themselves not connecting to anything). As a result it is possible to get everywhere in the village by walking on roads
- Roads also all connect properly while house placement is random

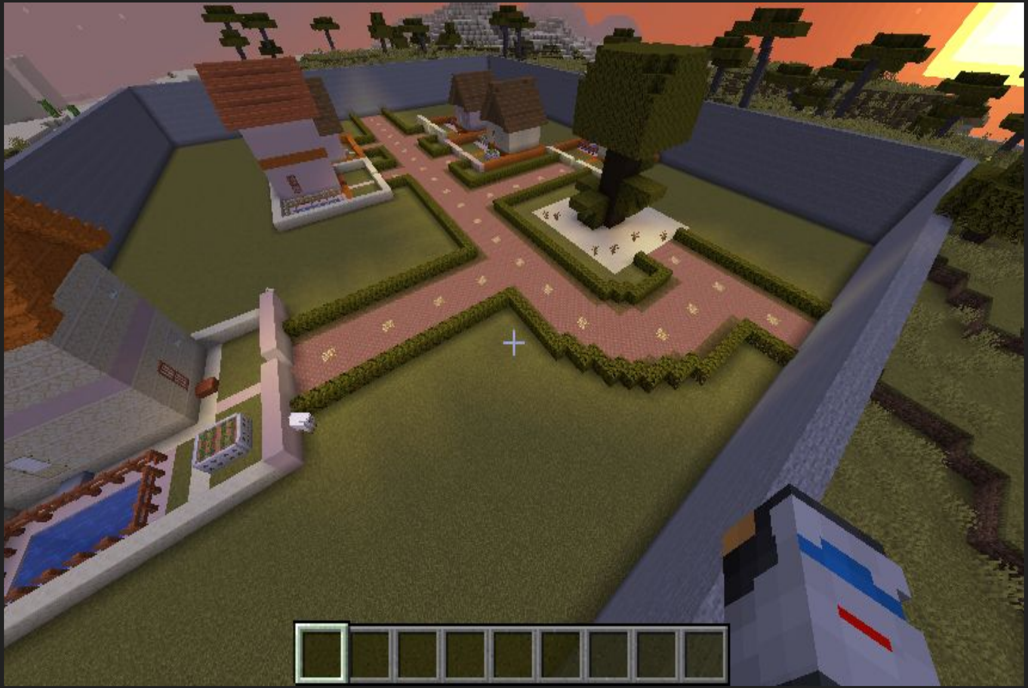
The village's roads are well-connected and exhibit complexity (e.g. they account for random house placement / they connect 3+ houses at different heights / the roads are curved in the (x, z) plane)

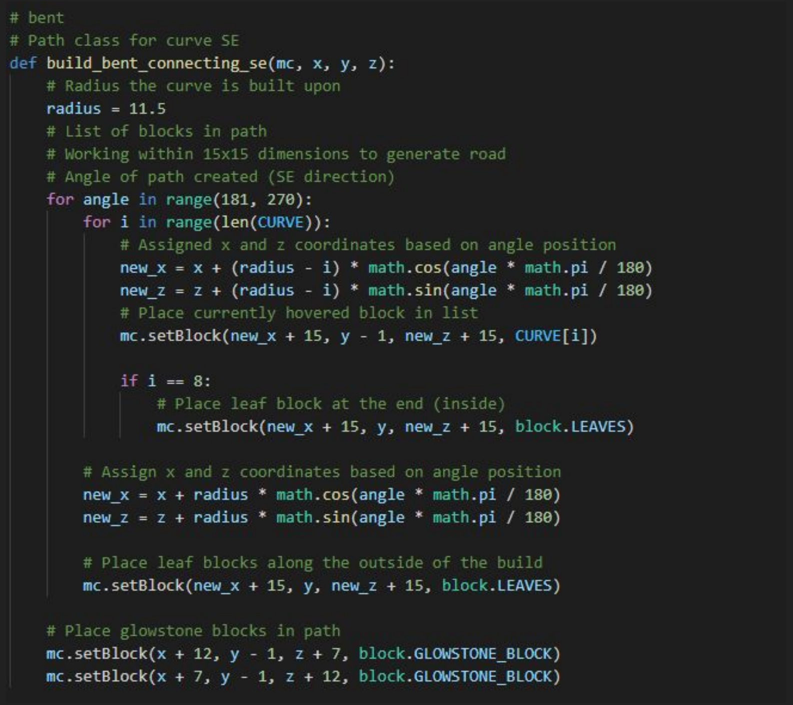




Roads

- Furthermore, roads exhibit an extra level of complexity by curving in the (x, z) plane via utilising trigonometric equations



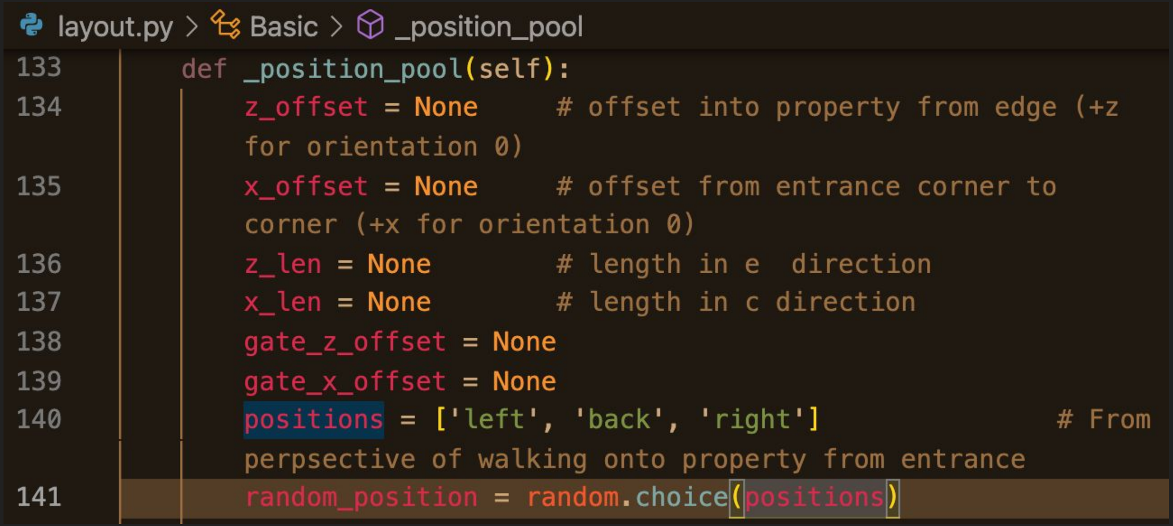


```
# bent
# Path class for curve SE
def build_bent_connecting_se(mc, x, y, z):
    # Radius the curve is built upon
    radius = 11.5
    # List of blocks in path
    # Working within 15x15 dimensions to generate road
    # Angle of path created (SE direction)
    for angle in range(181, 270):
        for i in range(len(CURVE)):
            # Assigned x and z coordinates based on angle position
            new_x = x + (radius - i) * math.cos(angle * math.pi / 180)
            new_z = z + (radius - i) * math.sin(angle * math.pi / 180)
            # Place currently hovered block in list
            mc.setBlock(new_x + 15, y - 1, new_z + 15, CURVE[i])

            if i == 8:
                # Place leaf block at the end (inside)
                mc.setBlock(new_x + 15, y, new_z + 15, block.LEAVES)

        # Assign x and z coordinates based on angle position
        new_x = x + radius * math.cos(angle * math.pi / 180)
        new_z = z + radius * math.sin(angle * math.pi / 180)

        # Place leaf blocks along the outside of the build
        mc.setBlock(new_x + 15, y, new_z + 15, block.LEAVES)

    # Place glowstone blocks in path
    mc.setBlock(x + 12, y - 1, z + 7, block.GLOWSTONE_BLOCK)
    mc.setBlock(x + 7, y - 1, z + 12, block.GLOWSTONE_BLOCK)
```

Swimming Pool

Jin

- Possible positions (relative to property entrance):
  - Back
  - Left
  - Right



layout.py > Basic > _position_pool

```python
    def _position_pool(self):
        z_offset = None     # offset into property from edge (+z for orientation 0)
        x_offset = None     # offset from entrance corner to corner (+x for orientation 0)
        z_len = None        # length in e  direction
        x_len = None        # length in c direction
        gate_z_offset = None
        gate_x_offset = None
        positions = ['left', 'back', 'right']               # From perpsective of walking onto property from entrance
        random_position = random.choice(positions)
```

Randomisation

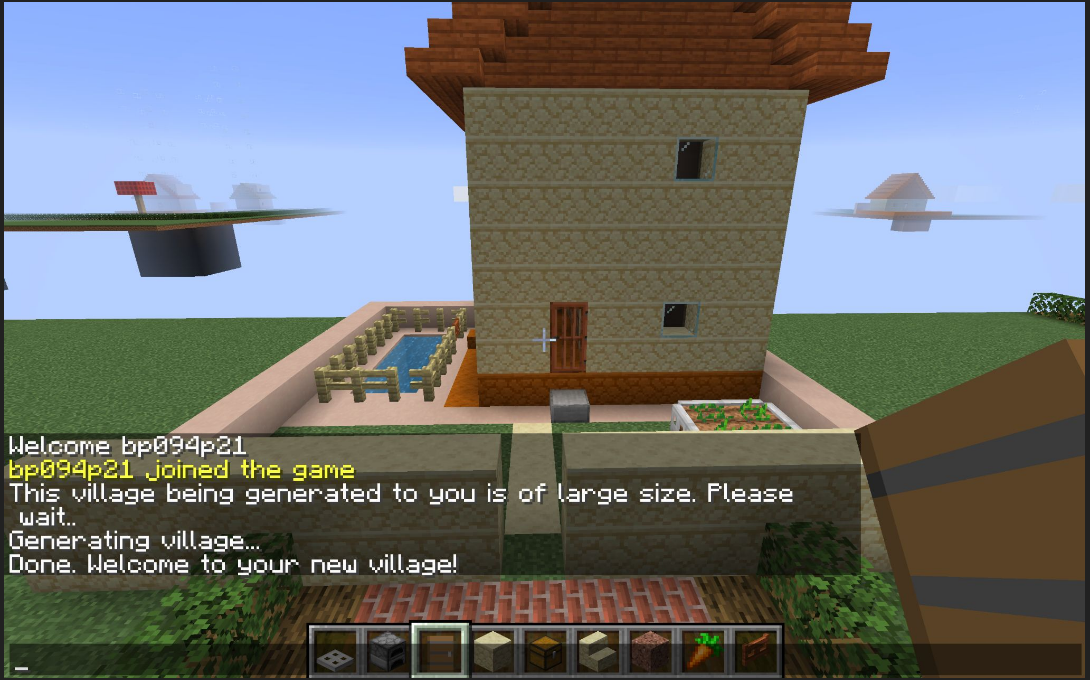
Jin

- Component positioning
- House dimensions
- Single/Double story
- Outdoor Feature presence/type
- Block types

Randomisation

Jin

- Component positioning

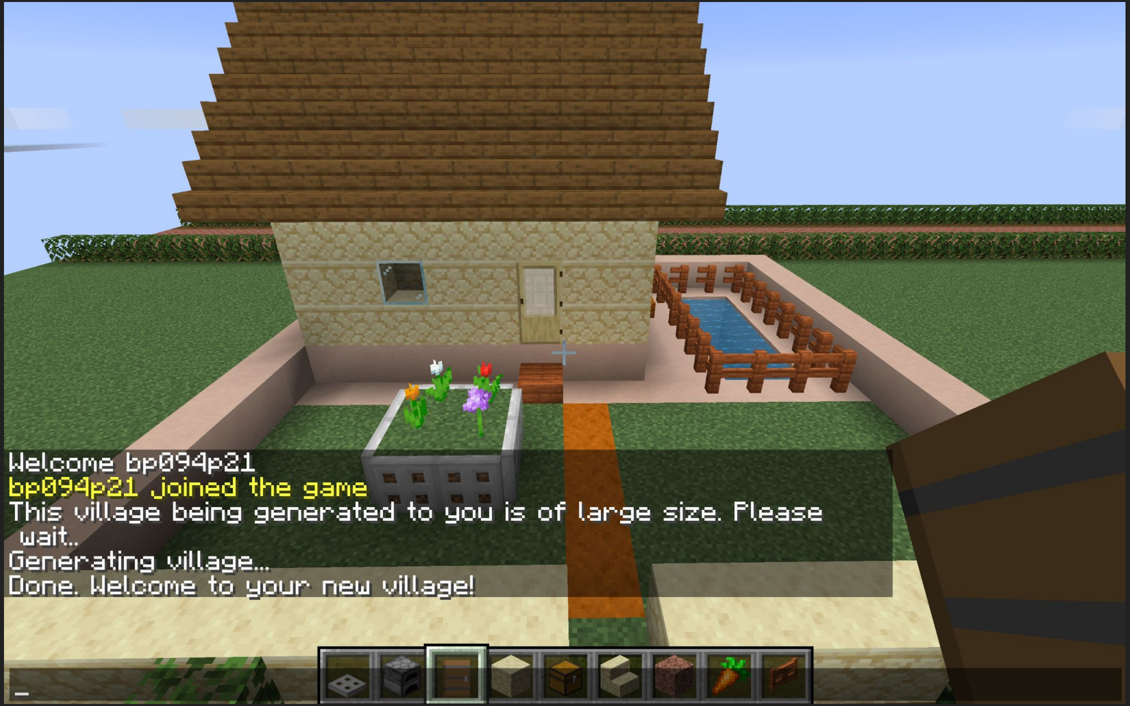


```
Welcome bp094p21
bp094p21 joined the game
This village being generated to you is of large size. Please
 wait..
Generating village...
Done. Welcome to your new village!
```

Randomisation

Jin

- Component positioning

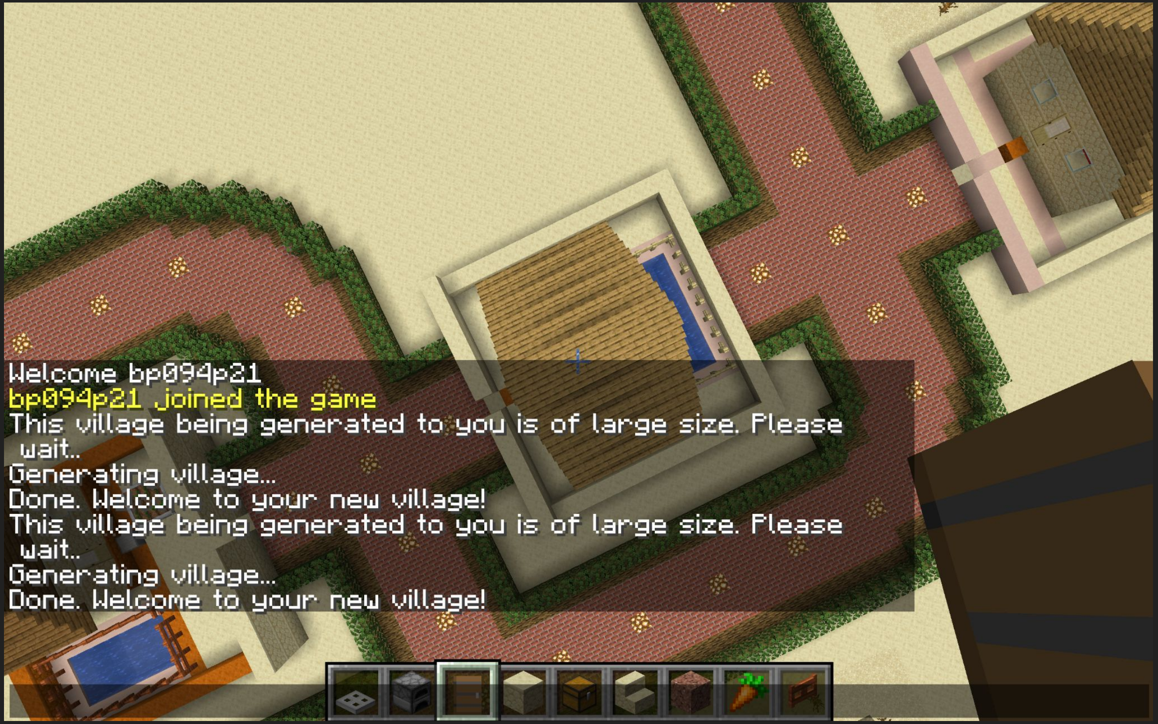


Welcome bp094p21
bp094p21 joined the game
This village being generated to you is of large size. Please wait..
Generating village...
Done. Welcome to your new village!

Randomisation

Jin

- Component positioning

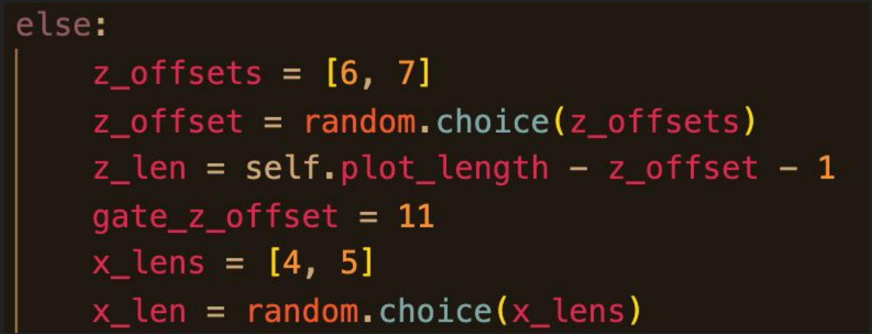


```
Welcome bp094p21
bp094p21 joined the game
This village being generated to you is of large size. Please
 wait..
Generating village...
Done. Welcome to your new village!
This village being generated to you is of large size. Please
 wait..
Generating village...
Done. Welcome to your new village!
```

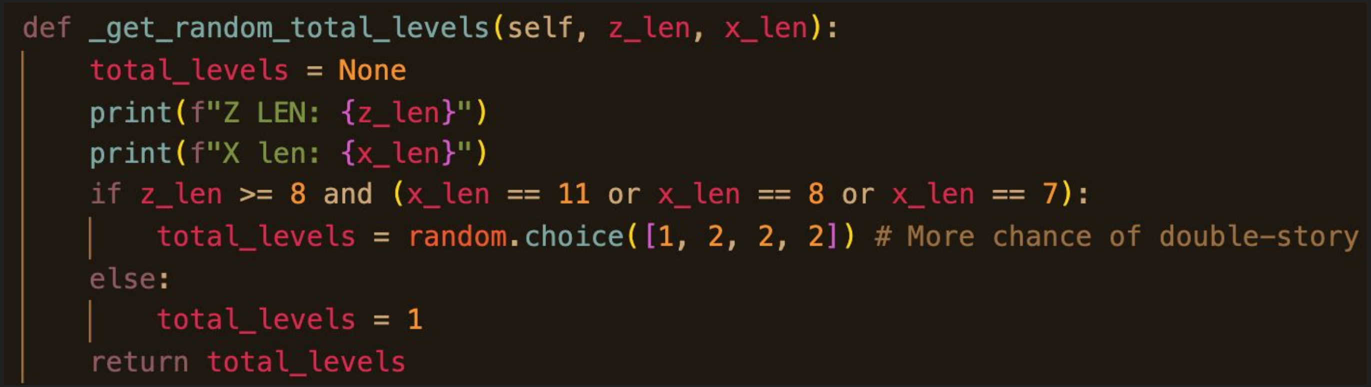
Randomisation

Jin

- House dimensions
- Single/Double story



```
else:
    z_offsets = [6, 7]
    z_offset = random.choice(z_offsets)
    z_len = self.plot_length - z_offset - 1
    gate_z_offset = 11
    x_lens = [4, 5]
    x_len = random.choice(x_lens)
```



```
def _get_random_total_levels(self, z_len, x_len):
    total_levels = None
    print(f"Z LEN: {z_len}")
    print(f"X len: {x_len}")
    if z_len >= 8 and (x_len == 11 or x_len == 8 or x_len == 7):
        total_levels = random.choice([1, 2, 2, 2]) # More chance of double-story
    else:
        total_levels = 1
    return total_levels
```

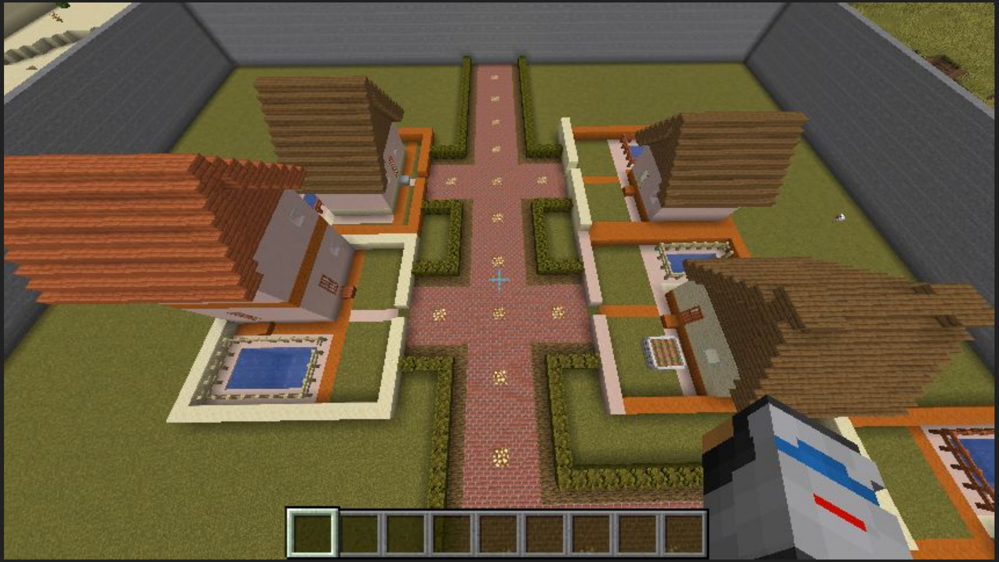
Randomisation

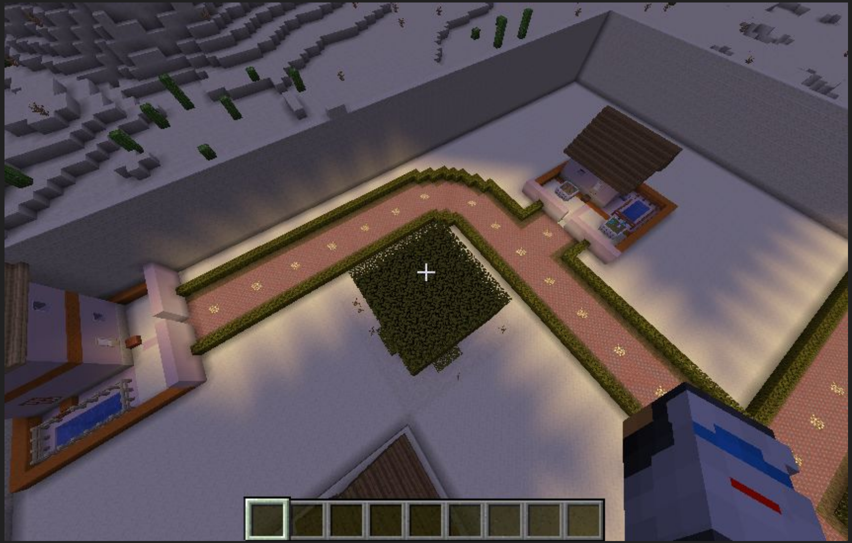
Lachie

- As noted earlier, the placement and structure of houses are random, as a result the paths need to connect to different placement of houses

Randomisation is used in the placement and structure of houses that makes the connectivity of roads / different rooms in the houses difficult to implement

Randomisation

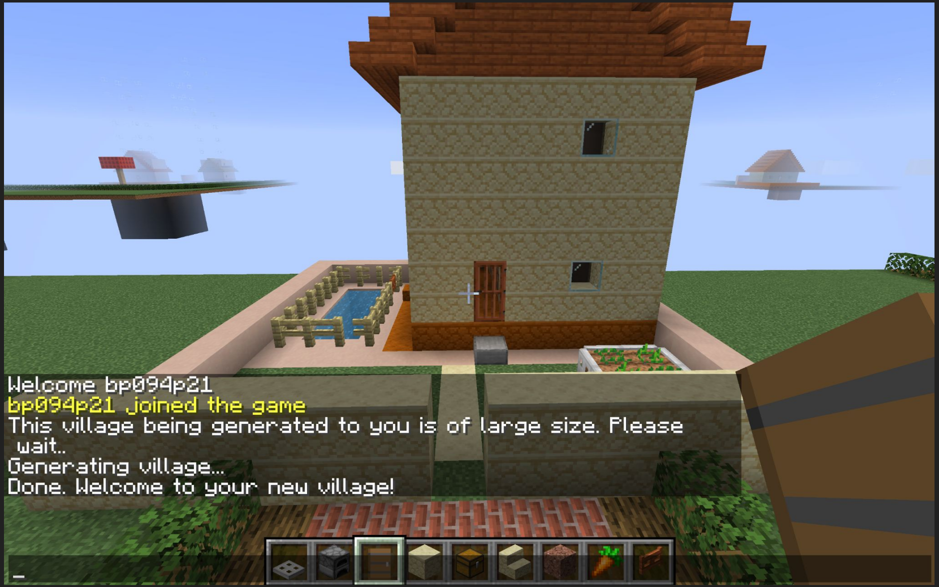




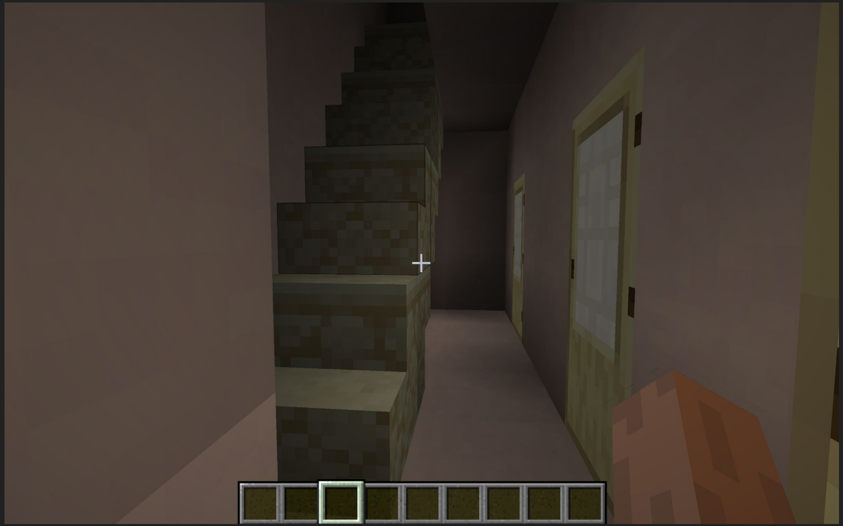
Additional Advanced Features

Jin

- Multi-story



Welcome bp094p21
bp094p21 joined the game
This village being generated to you is of large size. Please wait..
Generating village...
Done. Welcome to your new village!



Village Aesthetics and Creativity

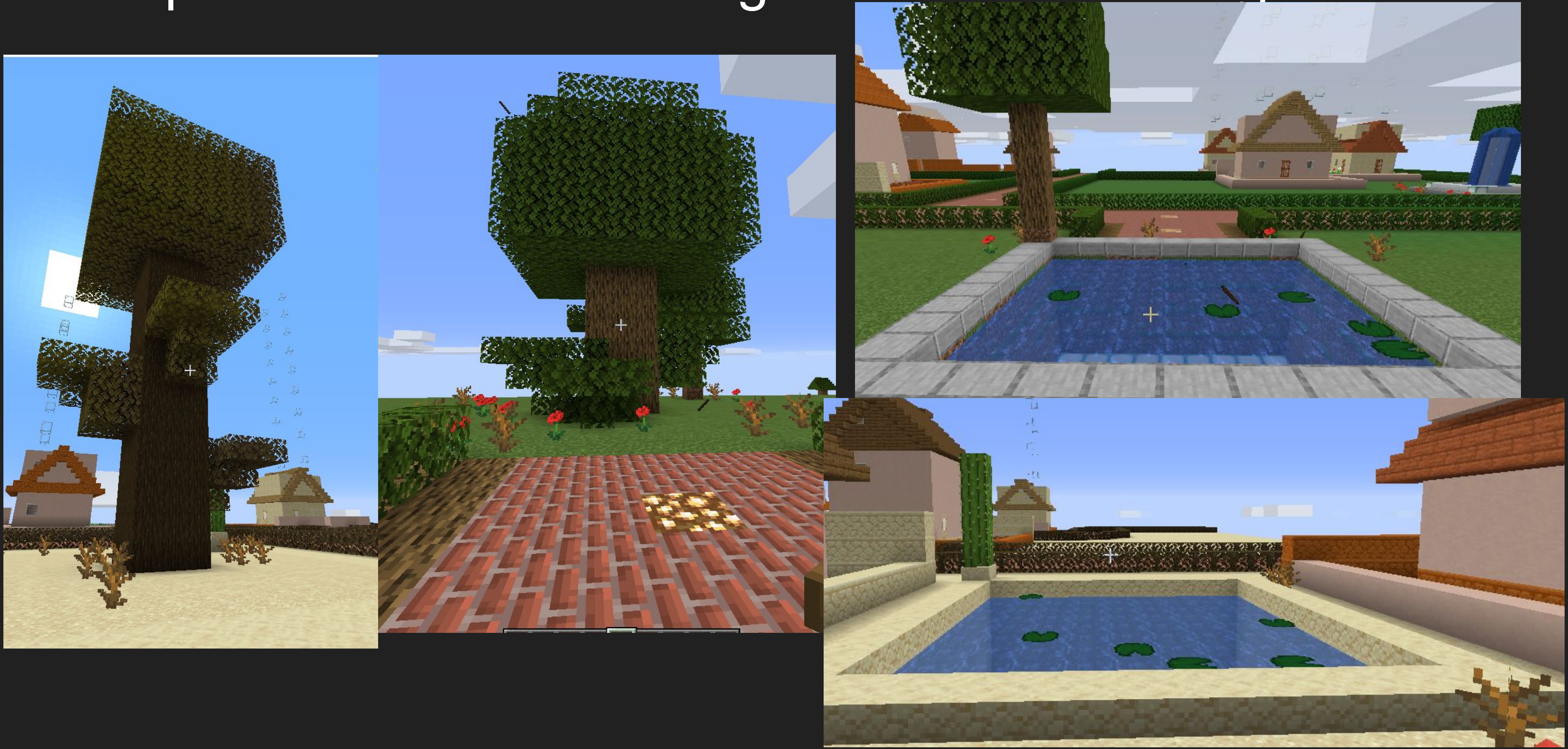
Roy

To bring more ‘life’ to the village, it was decided we would place in various structures for our aesthetics

These structures include:
-art features
-objects you would find at a park
-nature features

These structures would have their blocks change between the biomes they are placed in, as well as have a slight degree of randomisation to them, as shown in the tree structure and placements of the smaller trees in other structures

Examples of randomised height and biome change



Flexible Build Location

Roy

The village's spawn location is determined by the player's position at runtime, with buffering to ensure that buildings are not cut off if the player is at the edge of the game world.

The village builder program satisfies the constraint of ‘flexible build location’ , as it allows builds the village in an area around the players position and flattens the terrain if the area is not suitable for generation. Areas above a certain height are not deemed suitable for generation as there is likely not enough support for the village(mountain peaks), therefore the village will not generate a village at the programs height limit

Use of OOP Style

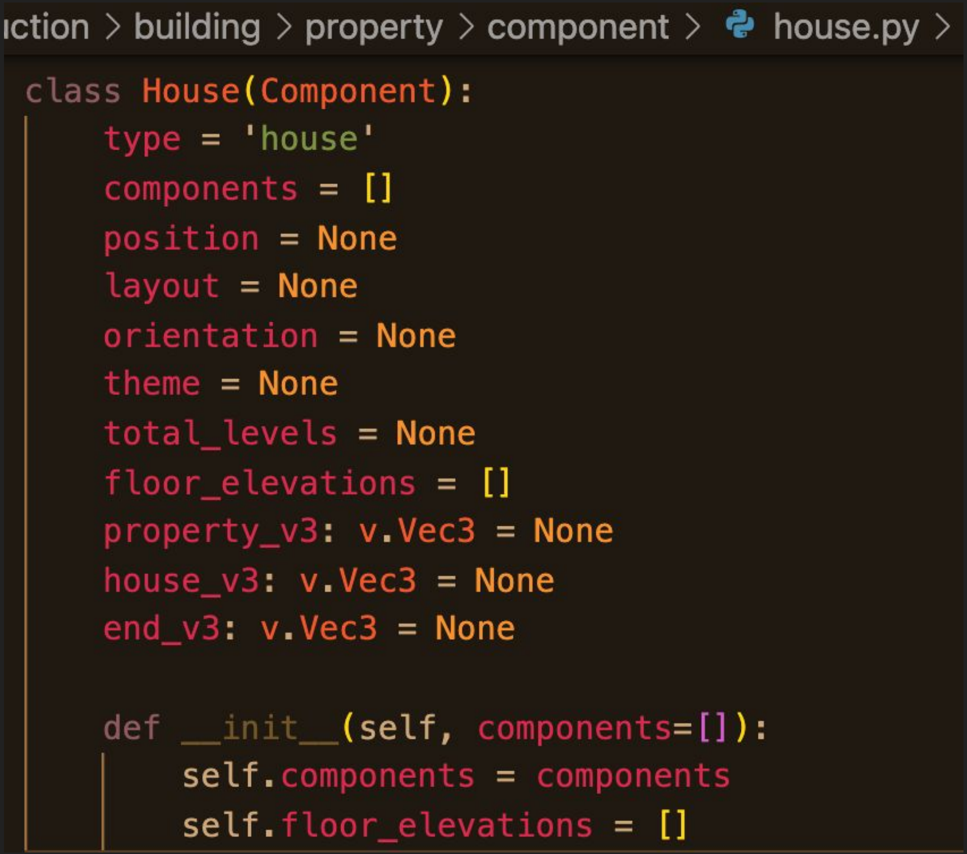
Jin

- The Property class contains a list of “Component” objects
    - Each item in the list is an instance of a different child class of “Component”

Use of OOP Style

Jin

- House Class



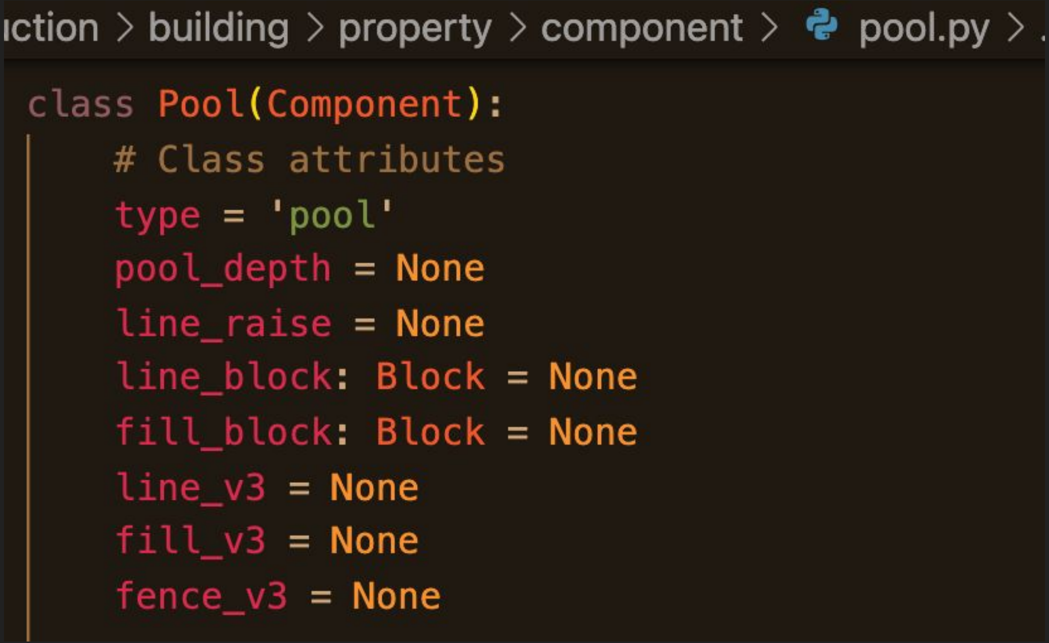
ıction > building > property > component > house.py >

```
class House(Component):
    type = 'house'
    components = []
    position = None
    layout = None
    orientation = None
    theme = None
    total_levels = None
    floor_elevations = []
    property_v3: v.Vec3 = None
    house_v3: v.Vec3 = None
    end_v3: v.Vec3 = None

    def __init__(self, components=[]):
        self.components = components
        self.floor_elevations = []
```

Use of OOP Style

Jin

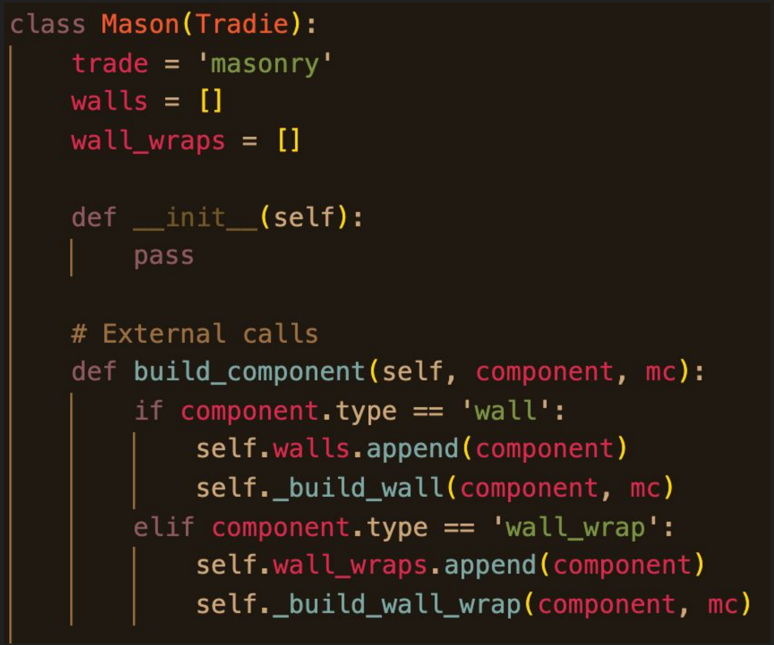
- Pool Class



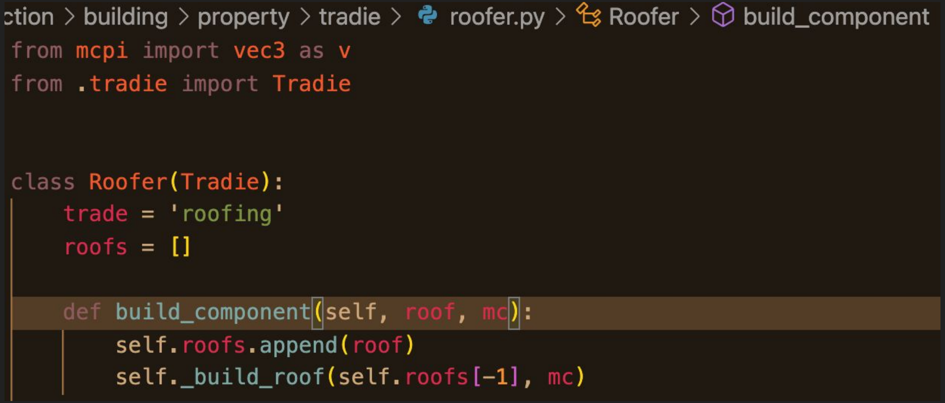
```
class Pool(Component):
    # Class attributes
    type = 'pool'
    pool_depth = None
    line_raise = None
    line_block: Block = None
    fill_block: Block = None
    line_v3 = None
    fill_v3 = None
    fence_v3 = None
```

Use of OOP Style

- Common function



```python
class Mason(Tradie):
    trade = 'masonry'
    walls = []
    wall_wraps = []

    def __init__(self):
        pass

    # External calls
    def build_component(self, component, mc):
        if component.type == 'wall':
            self.walls.append(component)
            self._build_wall(component, mc)
        elif component.type == 'wall_wrap':
            self.wall_wraps.append(component)
            self._build_wall_wrap(component, mc)
```

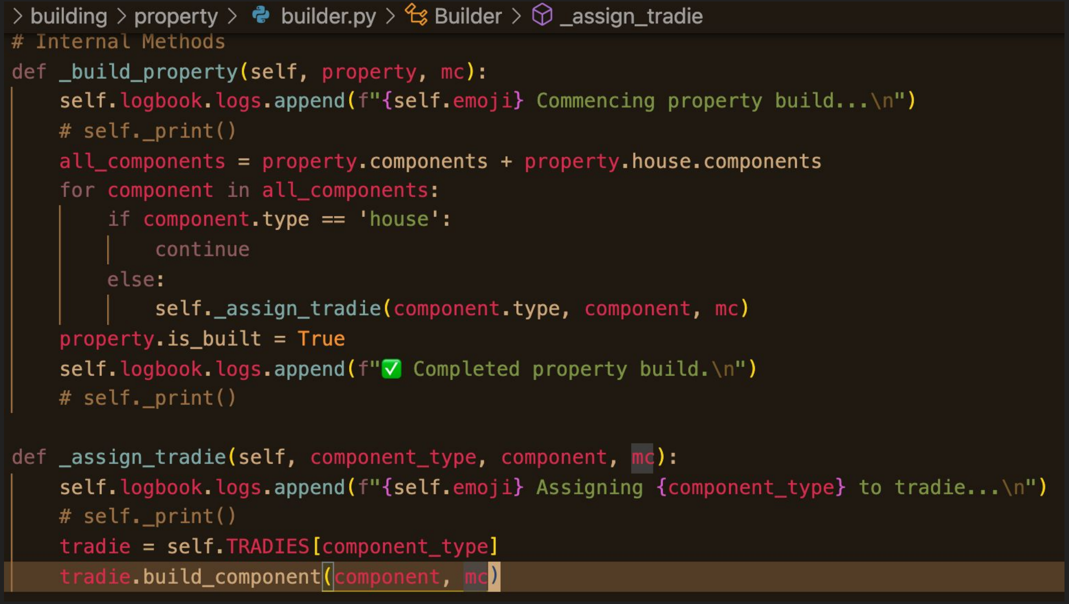


ction > building > property > tradie > roofer.py > Roofer > build_component

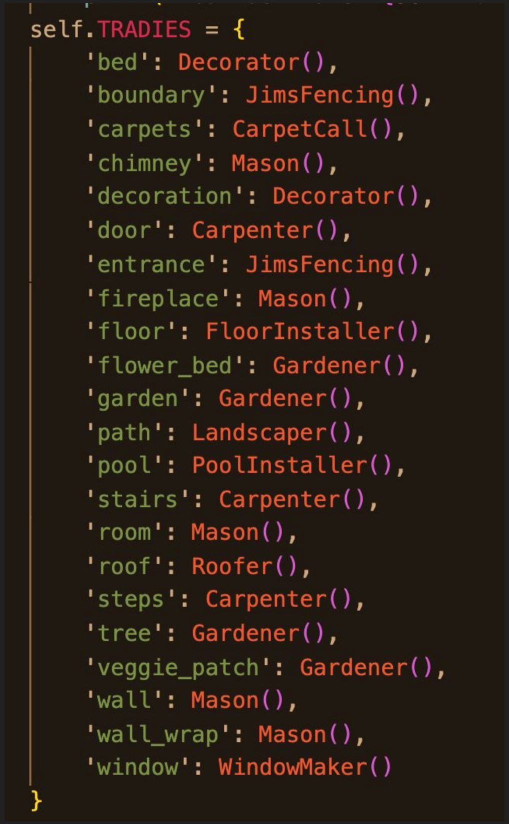
```python
from mcpi import vec3 as v
from .tradie import Tradie


class Roofer(Tradie):
    trade = 'roofing'
    roofs = []

    def build_component(self, roof, mc):
        self.roofs.append(roof)
        self._build_roof(self.roofs[-1], mc)
```

Use of OOP Style

- Builder: use for loop to assign tradie and build


> building > property > builder.py > Builder > _assign_tradie

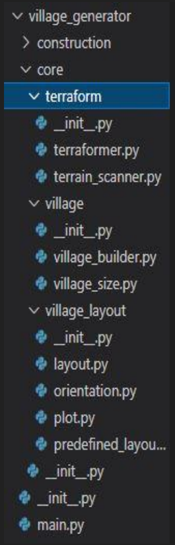
```python
# Internal Methods
def _build_property(self, property, mc):
    self.logbook.logs.append(f"{self.emoji} Commencing property build...\n")
    # self._print()
    all_components = property.components + property.house.components
    for component in all_components:
        if component.type == 'house':
            continue
        else:
            self._assign_tradie(component.type, component, mc)
    property.is_built = True
    self.logbook.logs.append(f"✅ Completed property build.\n")
    # self._print()

def _assign_tradie(self, component_type, component, mc):
    self.logbook.logs.append(f"{self.emoji} Assigning {component_type} to tradie...\n")
    # self._print()
    tradie = self.TRADIES[component_type]
    tradie.build_component(component, mc)
```



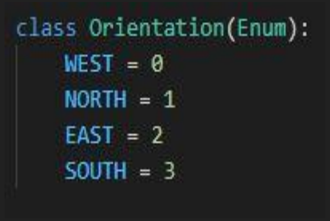
```python
self.TRADIES = {
    'bed': Decorator(),
    'boundary': JimsFencing(),
    'carpets': CarpetCall(),
    'chimney': Mason(),
    'decoration': Decorator(),
    'door': Carpenter(),
    'entrance': JimsFencing(),
    'fireplace': Mason(),
    'floor': FloorInstaller(),
    'flower_bed': Gardener(),
    'garden': Gardener(),
    'path': Landscaper(),
    'pool': PoolInstaller(),
    'stairs': Carpenter(),
    'room': Mason(),
    'roof': Roofer(),
    'steps': Carpenter(),
    'tree': Gardener(),
    'veggie_patch': Gardener(),
    'wall': Mason(),
    'wall_wrap': Mason(),
    'window': WindowMaker()
}
```

Good Programming Practices #1

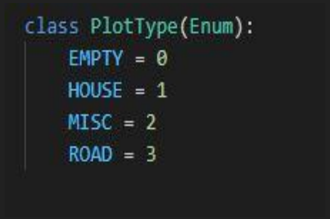
Modularity & Avoidance of Magic Numbers



```
v village_generator
  > construction
  v core
    v terraform
        __init__.py
        terraformer.py
        terrain_scanner.py
    v village
        __init__.py
        village_builder.py
        village_size.py
    v village_layout
        __init__.py
        layout.py
        orientation.py
        plot.py
        predefined_layou...
      __init__.py
    __init__.py
    main.py
```



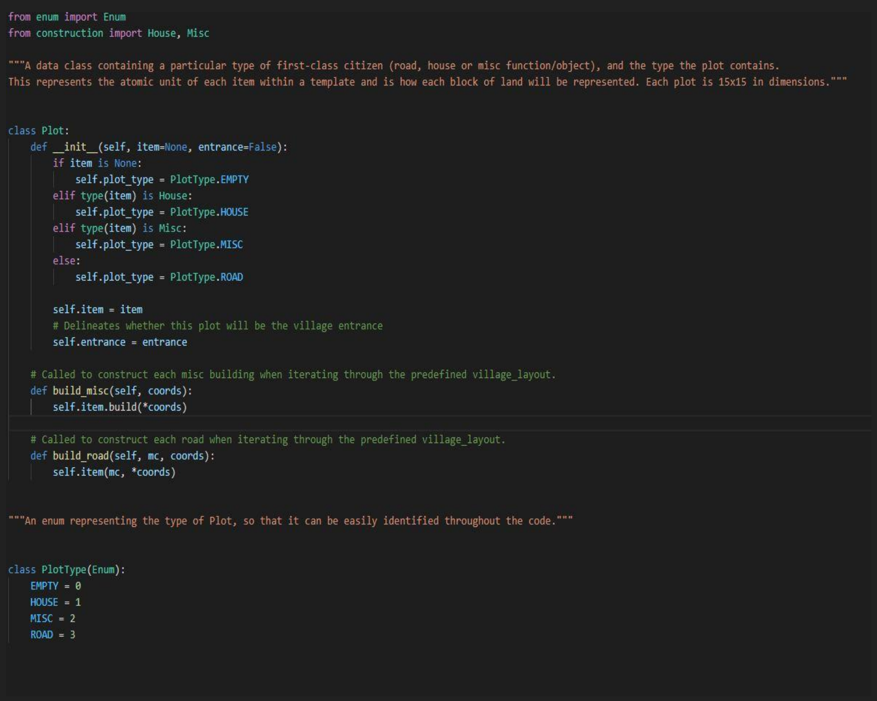
```python
class Orientation(Enum):
    WEST = 0
    NORTH = 1
    EAST = 2
    SOUTH = 3
```



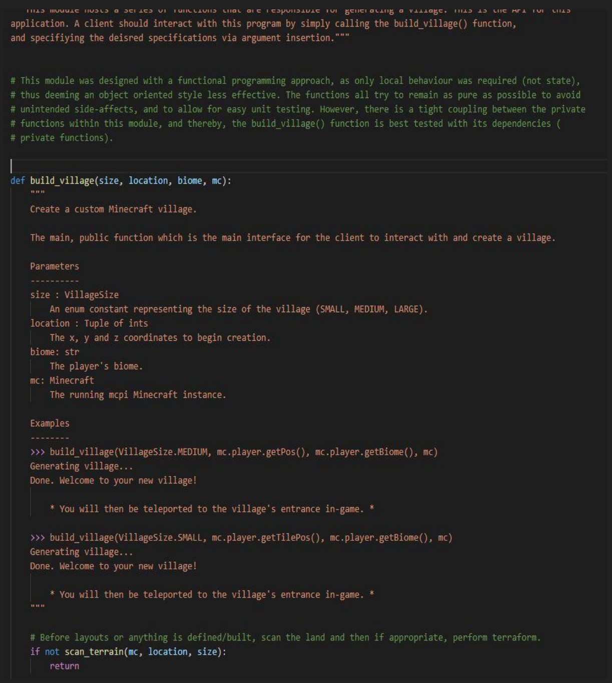
```python
class PlotType(Enum):
    EMPTY = 0
    HOUSE = 1
    MISC = 2
    ROAD = 3
```

Good Programming Practices #2

Formatting & Whitespace



```python
from enum import Enum
from construction import House, Misc

"""A data class containing a particular type of first-class citizen (road, house or misc function/object), and the type the plot contains.
This represents the atomic unit of each item within a template and is how each block of land will be represented. Each plot is 15x15 in dimensions."""


class Plot:
    def __init__(self, item=None, entrance=False):
        if item is None:
            self.plot_type = PlotType.EMPTY
        elif type(item) is House:
            self.plot_type = PlotType.HOUSE
        elif type(item) is Misc:
            self.plot_type = PlotType.MISC
        else:
            self.plot_type = PlotType.ROAD

        self.item = item
        # Delineates whether this plot will be the village entrance
        self.entrance = entrance

    # Called to construct each misc building when iterating through the predefined village_layout.
    def build_misc(self, coords):
        self.item.build(*coords)

    # Called to construct each road when iterating through the predefined village_layout.
    def build_road(self, mc, coords):
        self.item(mc, *coords)


"""An enum representing the type of Plot, so that it can be easily identified throughout the code."""


class PlotType(Enum):
    EMPTY = 0
    HOUSE = 1
    MISC = 2
    ROAD = 3
```



```python
    This module hosts a series of functions that are responsible for generating a village. This is the API for this
application. A client should interact with this program by simply calling the build_village() function,
and specifiying the deisred specifications via argument insertion."""


# This module was designed with a functional programming approach, as only local behaviour was required (not state),
# thus deeming an object oriented style less effective. The functions all try to remain as pure as possible to avoid
# unintended side-affects, and to allow for easy unit testing. However, there is a tight coupling between the private
# functions within this module, and thereby, the build_village() function is best tested with its dependencies (
# private functions).


def build_village(size, location, biome, mc):
    """
    Create a custom Minecraft village.

    The main, public function which is the main interface for the client to interact with and create a village.

    Parameters
    ----------
    size : VillageSize
        An enum constant representing the size of the village (SMALL, MEDIUM, LARGE).
    location : Tuple of ints
        The x, y and z coordinates to begin creation.
    biome: str
        The player's biome.
    mc: Minecraft
        The running mcpi Minecraft instance.

    Examples
    --------
    >>> build_village(VillageSize.MEDIUM, mc.player.getPos(), mc.player.getBiome(), mc)
    Generating village...
    Done. Welcome to your new village!

        * You will then be teleported to the village's entrance in-game. *

    >>> build_village(VillageSize.SMALL, mc.player.getTilePos(), mc.player.getBiome(), mc)
    Generating village...
    Done. Welcome to your new village!

        * You will then be teleported to the village's entrance in-game. *
    """

    # Before layouts or anything is defined/built, scan the land and then if appropriate, perform terraform.
    if not scan_terrain(mc, location, size):
        return
```