Decomposition
Class outline:

- Modules
- Packages
- Modularity
- Modular design
Modules
Python modules

A Python module is a file typically containing function or class definitions.

link.py:

class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' ' 
            self = self.rest
        return string + str(self.first) + '>'
Importing

Importing a whole module:

```python
import link

ll = link.Link(3, link.Link(4, link.Link(5)))
```

Importing specific names:

```python
from link import Link

ll = Link(3, Link(4, Link(5)))
```

Importing all names:

```python
from link import *

ll = Link(3, Link(4, Link(5)))
```
Importing with alias

I don't recommend aliasing a class or function name:

```python
from link import Link as LL
ll = LL(3, LL(4, LL(5)))
```

But aliasing a whole module is sometimes okay (and is common in data science):

```python
import numpy as np
b = np.array([[1.5, 2, 3], [4, 5, 6]])
```
Running a module

This command runs a module:

```
python module.py
```

When run like that, Python sets a global variable `__name__` to "main". That means you often see code at the bottom of modules like this:

```
if __name__ == "__main__":
    # use the code in the module somehow
```

The code inside that condition will be executed as well, but only when the module is run directly.
Packages
Python packages

A **Python package** is a way of bundling multiple related modules together. Popular packages are NumPy and Pillow.

Example package structure:

```plaintext
sound/
  __init__.py
  formats/
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
...
  effects/
    __init__.py
    echo.py
    surround.py
    reverse.py
...
  filters/
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
...
```

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sound/</td>
<td>Top-level package</td>
</tr>
<tr>
<td></td>
<td>Initialize the sound package</td>
</tr>
<tr>
<td></td>
<td>Subpackage for file format conversions</td>
</tr>
<tr>
<td>formats/</td>
<td>Subpackage for file format conversions</td>
</tr>
<tr>
<td>effects/</td>
<td>Subpackage for sound effects</td>
</tr>
<tr>
<td>filters/</td>
<td>Subpackage for filters</td>
</tr>
</tbody>
</table>
Importing from a package

Importing a whole path:

```python
import sound.effects.echo

sound.effects.echo.echofilter(input, output, delay=0.7)
```

Importing a module from the path:

```python
from sound.effects import echo
echo.echofilter(input, output, delay=0.7, atten=4)
```
Installing packages

The Python Package Index is a repository of packages for the Python language.

Once you find a package you like, `pip` is the standard way to install:

```
$ pip install nltk
```

You may need to use `pip3` if your system defaults to Python 2.
Modularity
Modular design

A design principle: Isolate different parts of a program that address different concerns.

A modular component can be developed and tested independently.

Ways to isolate in Python:
Modular design

A design principle: Isolate different parts of a program that address different concerns.

A modular component can be developed and tested independently.

Ways to isolate in Python:

- Functions
- Classes
- Modules
- Packages
Hog design

- **hog.py**: Strategies
- **dice.py**: Dice rolls

---

hog_gui.py

Plays the game according to rules.
Ants design

ants.py

<table>
<thead>
<tr>
<th>Place</th>
<th>Insect</th>
<th>GameState</th>
<th>AssaultPlan</th>
</tr>
</thead>
</table>

Defines classes for all game entities.

ants_strategies.py

<table>
<thead>
<tr>
<th>start_with_strategy</th>
</tr>
</thead>
</table>

A simple strategy.

gui.py

| GUI | HttpHandler | run |

Runs a browser-based game server.

See also: Ants class diagram
Icon project
Icon design

**icon.py**

- Icon
- Pixel
- Color

Defines classes for icon + parts.

**display_frame.py**

- DisplayFrame

Defines class for displaying icon on canvas.

**main.py**

Creates an Icon and displays it in DisplayFrame.
An OOP Icon

Goal: Use OOP to represent an Icon with pixels at a particular location with a particular color.

Icon

Pixel

width=8
height=8

x = 7  \quad r = 255
y = 0  \quad g = 0
\quad \quad b = 0
The Color class

class Color:

    def __init__(self, r, g, b):
        self.r = r
        self.g = g
        self.b = b

    def __repr__(self):
        return f"Color({self.r},{self.g},{self.b})"

    def to_hex(self):
        return f"#{{self.r:02x}}{{self.g:02x}}{{self.b:02x}}"

red = Color(255, 0, 0)
print(red.to_hex())
The Pixel class

class Pixel:
    def __init__(self, x, y, r, g, b):
        self.x = x
        self.y = y
        self.color = Color(r, g, b)

    def __repr__(self):
        return f"Pixel({self.x},{self.y},{self.color})"

pixel = Pixel(0, 7, 255, 0, 0)
print(pixel.color.to_hex())
The Icon class

class Icon:

    def __init__(self, width, height, pixels=None):
        self.width = width
        self.height = height
        self.pixels = pixels
        if not self.pixels:
            self.pixels = [Pixel(x, y, 0, 0, 0)
                           for x in range(width) for y in range(height)]

    def __repr__(self):
        pixels = ",".join([repr(pixel) for pixel in self.pixels])
        return f"Icon({self.width}, {self.height}, {self.pixels})"

icon = Icon(2, 2, [Pixel(0, 0, 255, 0, 0),
                    Pixel(0, 1, 255, 50, 0),
                    Pixel(1, 0, 255, 100, 0),
                    Pixel(1, 1, 255, 150, 0)])

for pixel in icon.pixels:
    pixel.color.g += 50
The DisplayFrame class

```python
from tkinter import Canvas, Frame, BOTH, font

class DisplayFrame(Frame):
    def __init__(self):
        super().__init__()
        self.pack(fill=BOTH, expand=1)
        self.canvas = Canvas(self)
        self.canvas.pack(fill=BOTH, expand=1)

    def draw_icon(self, icon):
        x_offset = 50
        y_offset = 50
        pixel_size = 20

        for pixel in icon.pixels:
            top_left_x = x_offset + pixel.x * pixel_size
            top_left_y = y_offset + pixel.y * pixel_size
            self.canvas.create_rectangle(
                top_left_x,
                top_left_y,
                top_left_x + pixel_size,
                top_left_y + pixel_size,
                outline="",
                fill=pixel.color.to_hex())
```
All together

```python
from tkinter import Tk

from icon import Icon, Pixel, Color
from display_frame import DisplayFrame

# Initialize the Tkinter frame and canvas
root = Tk()

display = DisplayFrame()
display.draw_icon(icon)

# Run Tkinter loop
root.mainloop()
```

Visit the Repl.it demo to see all the classes used with the Python tkinter package for graphics rendering.
Iterator-producing functions

What happens if we...

map the pixels?

```python
cchanger = lambda p: Pixel(p.x, p.y,
    p.x * 30,
    p.color.g + 30,
    p.y * 30)
icon.pixels = list(map(changer, icon.pixels))
```

filter the pixels?

```python
is_odd = lambda p: p.x % 2 == 0
icon.pixels = list(filter(is_odd, icon.pixels))
```
Iterable-processing functions

What happens if we ask for the min and max of the pixels?

```
max_pix = max(icon.pixels)
min_pix = min(icon.pixels)
```
Iterable-processing functions

What happens if we ask for the min and max of the pixels?

```python
max_pix = max(icon.pixels)
min_pix = min(icon.pixels)
```

Python doesn't know how to compare `Pixel` instances! Two options:

- Implement dunder methods (``__eq__``, ``__lt__``, etc)
- Pass in a key function that returns a numerical value:

```python
rgb_adder = lambda p: p.color.r + p.color.g + p.color.b
max_pix = max(icon.pixels, key=rgb_adder)
min_pix = min(icon.pixels, key=rgb_adder)
```
Python Project of The Day!
Panda3D

Panda3D: an open-source, completely free-to-use engine for realtime 3D games, visualizations, simulations, experiments. Written in C++ with Python bindings.

Github organization, Open Collective