Python for videogames

Videogames Technology
Asignatura transversal

Departamento de Automática

Universidad de Alcalá

Objectives

1. Understand the relevance to use modules and packages.
2. Install some widely used Python packages.
3. First contact with Arcade.

Bibliography

- The Python Tutorial. *Chapter 6: Modules*. (Link)
- Paul Vincent Craven. *Easy 2D game creation with Arcade*. (Link)
- Paul Vincent Craven. *Learn to Program with Arcade*. (Link)

Table of Contents

Introduction

Why modules?

- **Main function**: Organization.
- **Reuse**: To provide software solutions, that have been proven to work, to solve similar problems.

Using modules

Creation and Implementation

A module is just a Python script with `.py` extension

fibo.py

```
1  def fib(n):
2      """Print a Fibonacci series up to n """
3      a, b = 0, 1
4      while a < n:
5          print(a, end=' ')
6          a, b = b, a+b
7      print()
8
9  def fib2(n):
10     """Print a Fibonacci series up to n """
11     result = [] # Declare a new list
12     a, b = 0, 1
13     while a < n:
14         result.append(a) # Add to the list
15         a, b = b, a+b
16     return result
```

Using modules

How do I use them? (I)

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
>>> fib = fibo.fib
>>> fib(100)
1 1 2 3 5 13 21 34 55 89
```

Using modules

How do I use them? (II)

A module can import other modules

- Name conflicts may arise: Each module has a symbol table
- It means you should invoke it as `modname.itemname`

It is possible to import items directly

- `from module import name1, name2`
- `from module import *`
- It uses the global symbol table (no need to use the modname)

```
>>> from fibo import fib, fib2
>>> fib(100)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Using modules

How do I use them? (III)

List zip file contents (file.zip must exist. Open in read mode)

```
1  import zipfile
2
3  file = zipfile.ZipFile("file.zip", "r")
4
5  # list filenames
6  for name in file.namelist():
7      print(name)
8
9  # list file information
10 for info in file.infolist():
11     print(info.filename, info.date_time, info.file_size)
```

Several examples here: `http://pymotw.com/2/PyMOTW-1.132.pdf`

Executing modules

Modules as scripts (I)

When a module is imported, its statements are executed

- It declares functions, classes, variables ...
- ... and also executes code
- It serves to initialize the module

Very useful to use modules as programs and libraries

Executing modules

Modules as scripts (II)

fibo2.py

```
1  def fib(n):
2      """Print a Fibonacci series up to n """
3      a, b = 0, 1
4      while a < n:
5          print(a, end=' ')
6          a, b = b, a+b
7      print()
8
9  if __name__ == "__main__":
10     import sys
11     fib(int(sys.argv[1]))
```

`__main__` is a special variable set to the module's name

```
    (In Linux console)
$ python3 fibo2.py 50
1 1 2 3 5 8 13 21 34
```

```
    (In Python interpreter)
>>> import fibo2
>>> fibo2.fib(50)
1 1 2 3 5 8 13 21 34
```

Executing modules

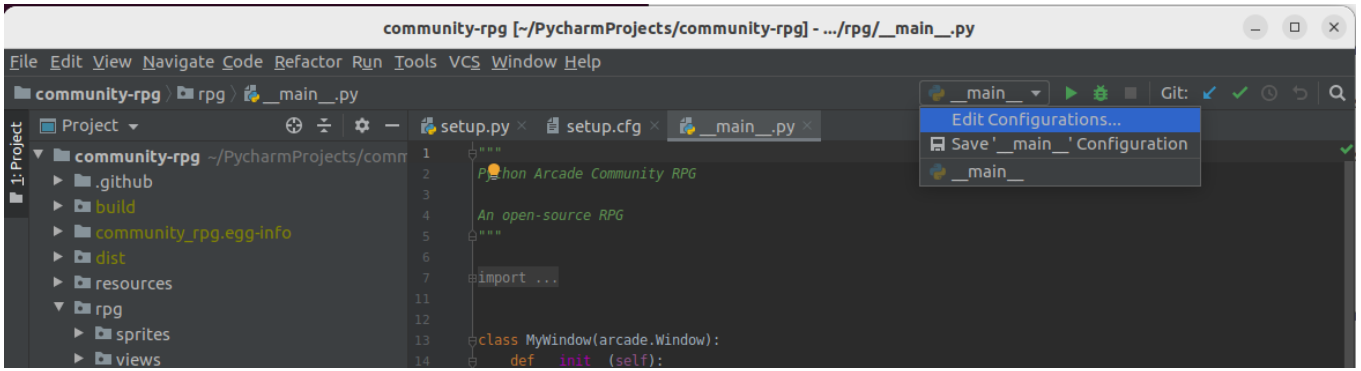
Modules as scripts (III)

The file `__main__.py` is an alternative entry point in a module

- Useful in command line mode and IDEs

```
(In Linux console)
$ python3 -m mymodule
```

IDEs allow the configuration of multiple entry points



Content of a module

The dir() function

`dir()`: Built-in function that returns the names defined in a module

- Without arguments, it returns your names

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir()
['__builtins__', ... , '__spec__']
>>> variable = 'Hello'
>>> dir()
['__builtins__', ... , '__spec__', 'variable']
```

Packages

Package concept (I)

If a module gets too big, many problems arise

- Name collisions
- It is good to organize modules in a bigger structure: *Packages*

Packages can be seen as "dotted module names"

- It is just a module that contains more modules
- Make life easier in big proyects
- The name `A.B` designates a submodule `B` in a package named `A`

Must contain an `__init__.py` file in the root directory

- Executed when the package is imported for the first time

Packages

Package concept (II)

Sound module structure

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Packages

Importing a package (I)

Importing an individual module

```
import sound.effects.echo
sound.effects.echo.echofilter(input, output)
```

Alternative way to import an individual module

```
from sound.effects import echo
echo.echofilter(input, output)
```

Alternative way to import an individual module

```
from sound.effects.echo import echofilter
echofilter(input, output)
```

Packages

Importing a package (II)

Imagine we run `from sound import *`

- In theory, it would import the whole package
- In practice, it would take too much time

There is a convention to avoid waste of resources

- There may be a variable `__all__` defined in `__init__`
- `__all__` contains modules to be imported

sounds/effects/__init__.py

```
__all__ = ["echo", "surround", "reverse"]
```

Packages

Installing packages (I)

Command-line automatic tool: `pip` (sometimes `pip3`)

- Very similar to `apt-get` in Linux

pip usage (from OS terminal)

```
$ python -m pip install SomePackage
```

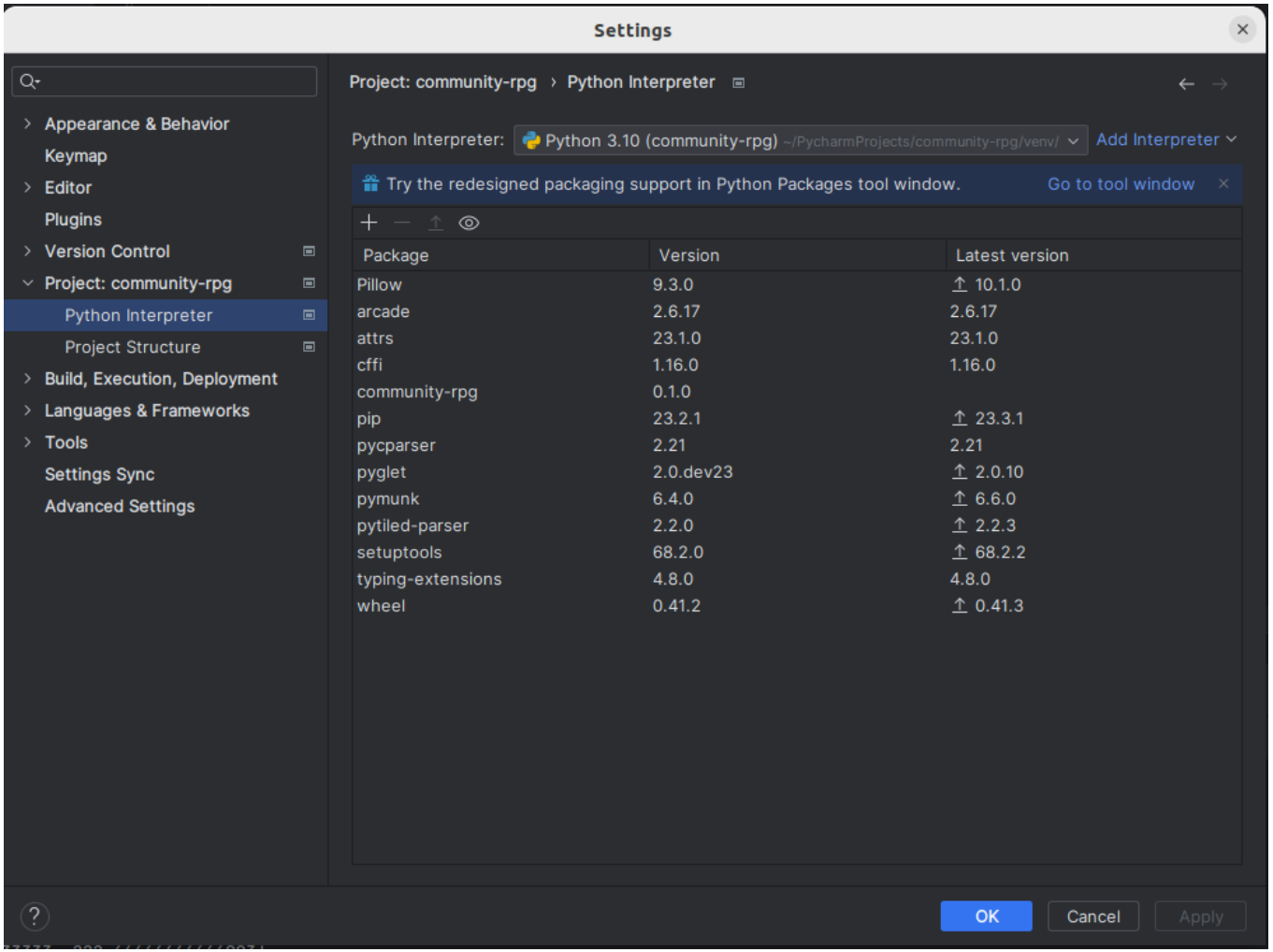
or

```
$ pip install SomePackage
```

```
$ pip install Pillow
```

Packages

Installing packages (II)



Virtual environments (I)

Versioning is problematic

- Python version (2.x, 3.x)
- Packages version

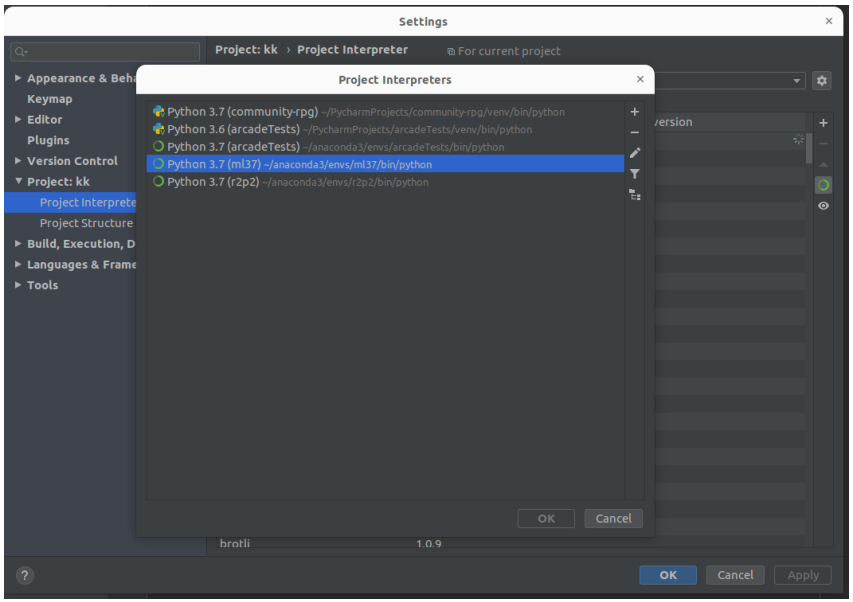
Solution: virtual environment

- Self-contained directory with a Python installation
- Particular version of Python and packages

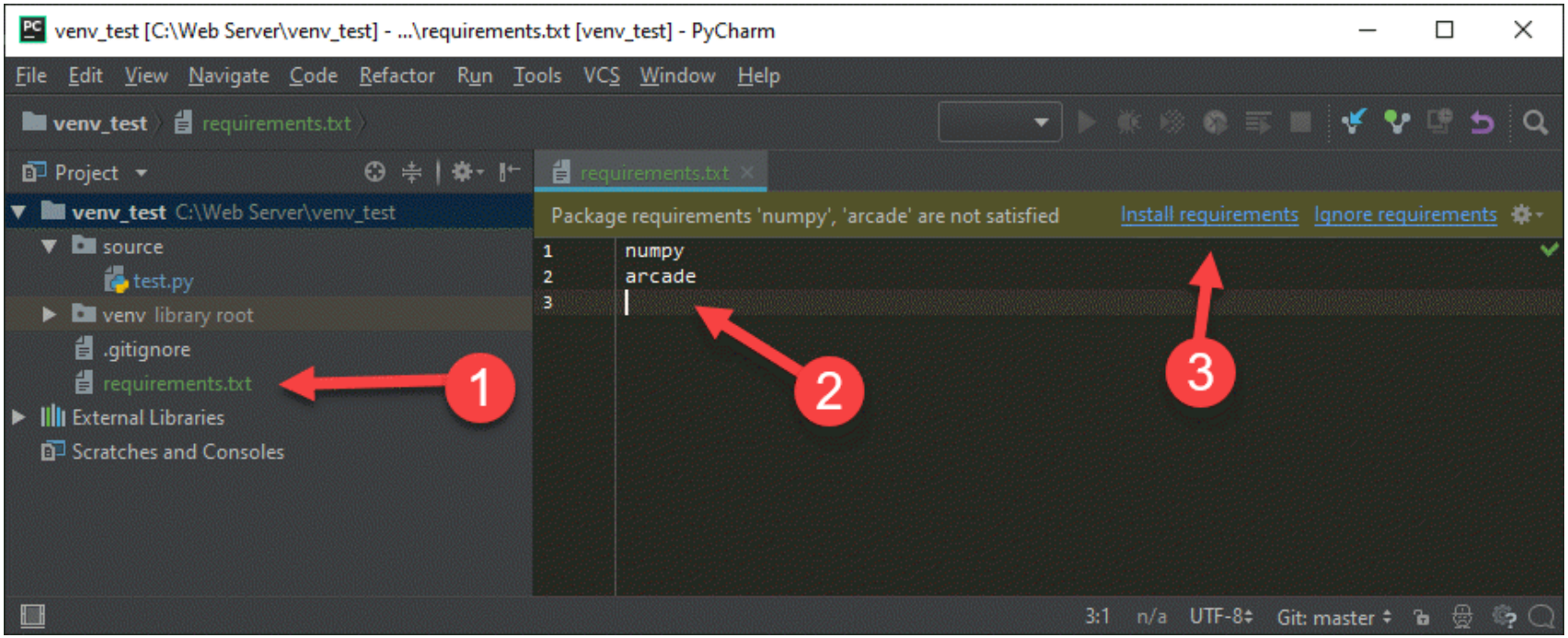
Different solutions

- venv
- conda

Great tool with `requirements.txt` or `setuptools`



Virtual environments (II)



(Source)

Cool code examples

Example 1: Open a web browser

browser.py

```
1 import webbrowser
2
3 url = input('Give me an URL: ')
4
5 webbrowser.open(url)
```

Cool code examples

Example 2: Create a thumbnail

thumbnail.py

```
1 from PIL import Image
2
3 size = (128, 128)
4 saved = "africa.jpg"
5
6 im = Image.open("africa.tif")
7 im.thumbnail(size)
8 im.save(saved)
9 im.show()
```

(Source)

africa.jpg

Cool code examples

Example 3: Send an email with Gmail

gmail.py

```
"""The first step is to create an SMTP object,
each object is used for connection
with one server."""

import smtplib
server = smtplib.SMTP('smtp.gmail.com', 587)

# Next, log in to the server
server.login("youremailusername", "password")

# Send the mail
msg = "\nHello!" # /n separates the message from the headers
server.sendmail("you@gmail.com", "target@example.com", msg)
```

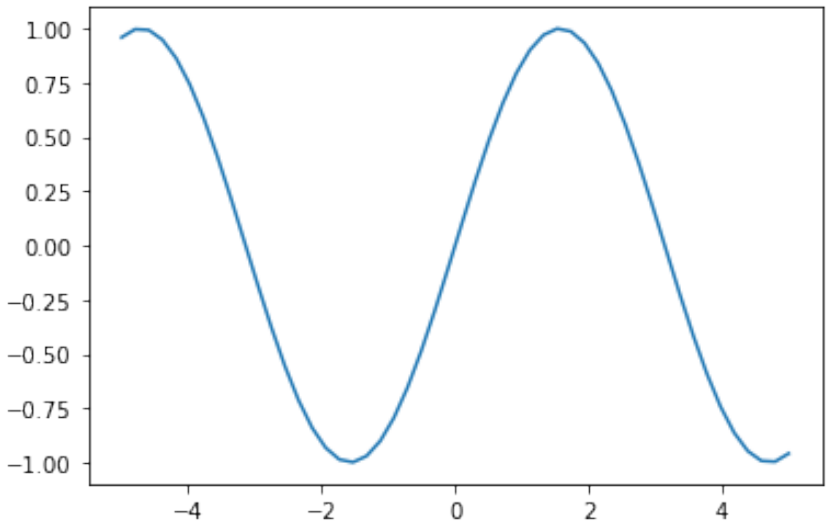
(Source)

Modules

Example 4: Plot

plot.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(-5, 5)
5 plt.plot(x, np.sin(x))
```

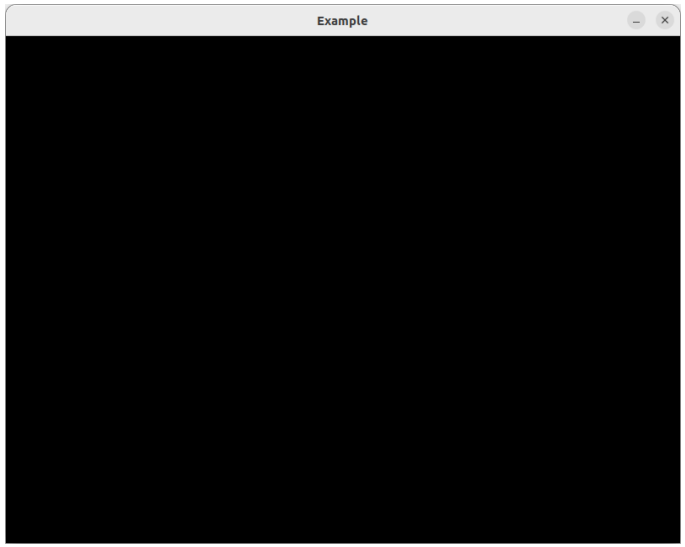


Modules

Example 5: Arcade

arcade.py

```
1 import arcade
2
3 WIDTH = 600
4 HEIGHT = 800
5
6 arcade.open_window(WIDTH, HEIGHT, "
      Example")
7
8 arcade.run()
```



- (API documentation)
- (Arcade source code)

Arcade

Introduction

Arcade is an easy-to-use 2D motor engine (i.e. a Python package)

- Created by Paul Vincent Craven
- Based on Python
- More or less painless game development
- Didactic
- Free software

Requires

- Python 3.6+
- OpenGL capable hardware

Dependences

- Pyglet - Mutimedia library for Python

(Arcade web site)

Arcade

Open a Window (I)

arcade.py

```
1 import arcade
2
3 WIDTH = 600
4 HEIGHT = 800
5
6 arcade.open_window(WIDTH, HEIGHT, "Example")
7
8 arcade.run()
```

Arcade

Drawing setup

drawing.py

```
1  import arcade
2
3  WIDTH = 600
4  HEIGHT = 800
5
6  arcade.open_window(WIDTH, HEIGHT,"Drawing Example")
7
8  arcade.set_background_color(arcade.color.WHITE)
9
10 arcade.start_render()
11
12 # Drawing here
13
14 arcade.finish_render()
15
16 arcade.run()
```

Arcade

Drawing (I)

(Source code)

Arcade

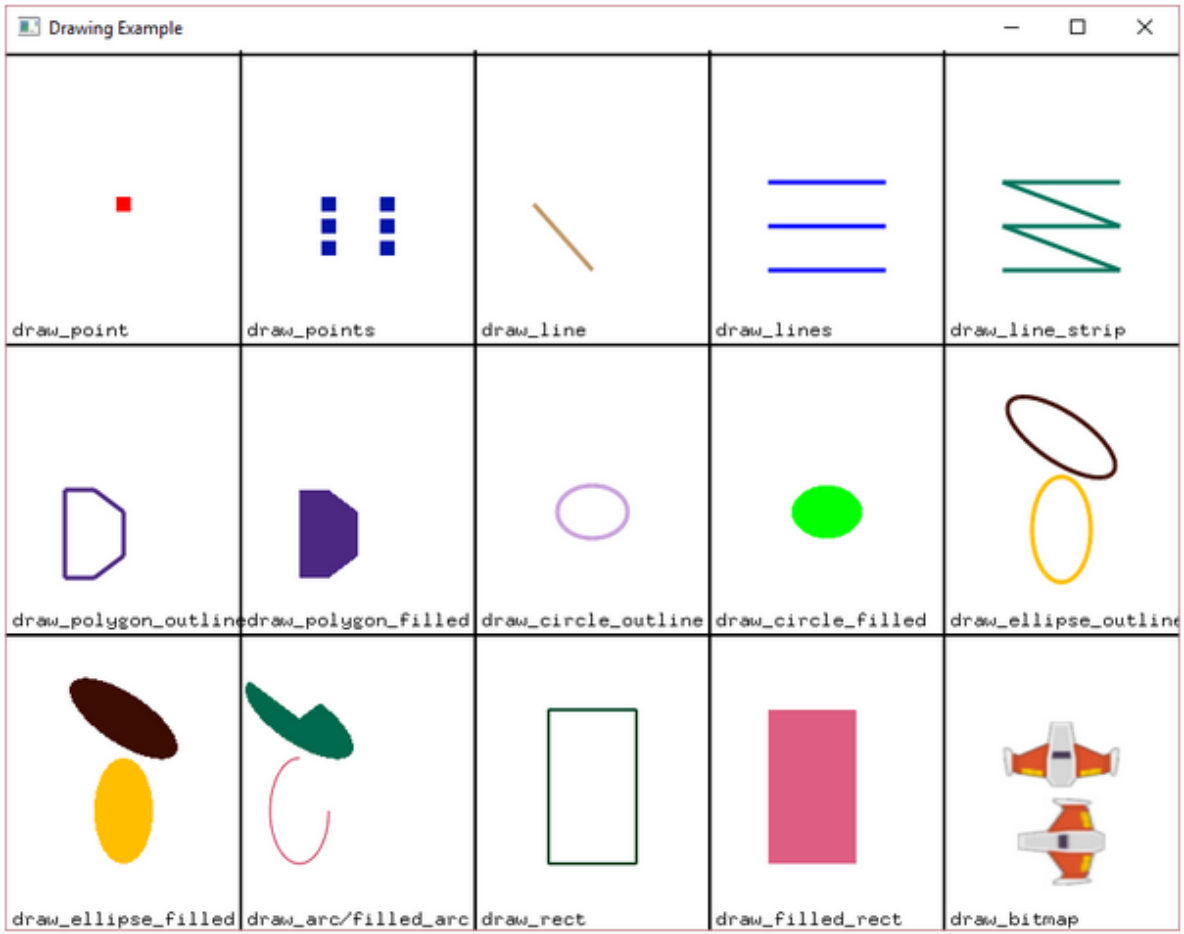
Drawing (II)

smile.py

```
# Draw the face
x = 300; y = 300; radius = 200
arcade.draw_circle_filled(x, y, radius, arcade.color.YELLOW)

# Draw the right eye
x = 370; y = 350; radius = 20
arcade.draw_circle_filled(x, y, radius, arcade.color.BLACK)

# Draw the left eye
x = 230; y = 350; radius = 20
arcade.draw_circle_filled(x, y, radius, arcade.color.BLACK)

# Draw the smile
x = 300; y = 280; width = 120; height = 100
start_angle = 190; end_angle = 350; line_width = 10
arcade.draw_arc_outline(x, y, width, height, arcade.color.BLACK,
                        start_angle, end_angle, line_width)
```

Arcade

Drawing primitives (I)



(Source code)

Arcade

Drawing primitives (II)

```
draw_rectangle_filled()
draw_rectangle_outline()
draw_lrtb_rectangle_filled()
draw_lrtb_rectangle_outline()
draw_xywh_rectangle_filled()
draw_xywh_rectangle_outline()

draw_polygon_filled()
draw_polygon_outline()

load_texture()
draw_texture_rectangle()
draw_xywh_rectangle_textured()

draw_triangle_filled()
draw_triangle_outline()
```

```
draw_arc_filled()
draw_arc_outline()

draw_circle_filled()
draw_circle_outline()

draw_ellipse_filled()
draw_ellipse_outline()

draw_line()
draw_line_strip()
draw_lines()

draw_point()
draw_points()
```

Arcade

Colors

How can I know which colors has Arcade available?

- The reference API is your friend!
- (`arcade.color` reference documentation)