Python for videogames

Videogames Technology
Asignatura transversal

Departamento de Automática

Universidad de Alcalá

Objectives

1. First contact with Arcade.
2. Understand modules and packages.
3. Basic package management.
4. Introduce virtual environments.

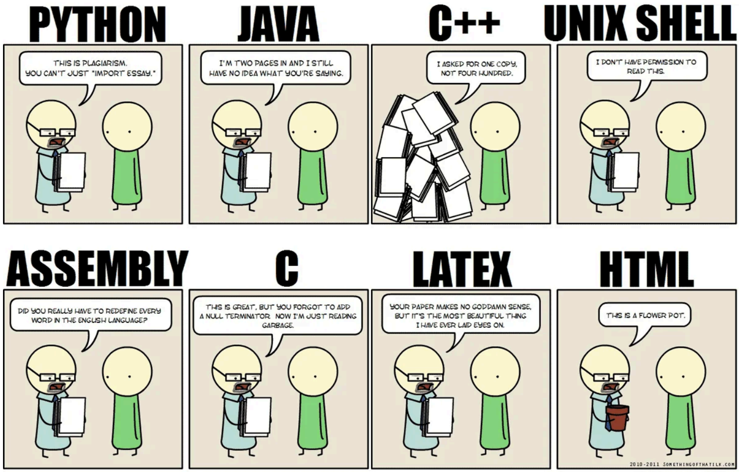
Bibliography

- 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

Python.Tips.Hub
How to do THING in PYTHON!!!
1morbillion likes

```
import thingDoerLibrary

thingDoerLibrary.DoThing()
```


PYTHON
THIS IS PLAGIARISM. YOU CAN'T JUST "IMPORT ESSAY."
JAVA
I'M TWO PAGES IN AND I STILL HAVE NO IDEA WHAT YOU'RE SAYING.
C++
I ASKED FOR ONE COPY, NOT FOUR HUNDRED.
UNIX SHELL
I DON'T HAVE PERMISSION TO READ THIS.
ASSEMBLY
DID YOU REALLY HAVE TO REDEFINE EVERY WORD IN THE ENGLISH LANGUAGE?
C
THIS IS GREAT, BUT YOU FORGOT TO ADD A NULL TERMINATOR. NOW I'M JUST READING GARBAGE.
LATEX
YOUR PAPER MAKES NO GODDAMN SENSE, BUT IT'S THE MOST BEAUTIFUL THING I HAVE EVER LAID EYES ON.
HTML
THIS IS A FLOWER POT.
2010-2011 SOMETHINGOFTHATILK.COM

Introduction

Why modules?

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

Modules

Creation

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

fibo.py

```
def fib(n):
    """Print a Fibonacci series up to n """
    a, b = 0, 1
    while a < n:
        print(a, end= ' ')
        a, b = b, a+b
    print()

def fib2(n):
    """Print a Fibonacci series up to n """
    result = [] # Declare a new list
    a, b = 0, 1
    while a < n:
        result.append(a) # Add to the list
        a, b = b, a+b
    return result
```

Modules

Using modules (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
```

Modules

Using modules (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
```

Modules

Using modules: Example

Modules usually are used as libraries

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

```
import zipfile

file = zipfile.ZipFile("file.zip", "r")

# list filenames
for name in file.namelist():
    print(name)

# list file information
for info in file.infolist():
    print(info.filename, info.date_time, info.file_size)
```

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

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

- Good to reuse code

Modules

Modules as scripts (II)

fibo2.py

```python
import sys

def fib(n):
    """Print a Fibonacci series up to n """
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

if __name__ == "__main__":
    fib(int(sys.argv[1]))
```

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

Linux/PowerShell console

```
$ python3 fibo2.py 50
1 1 2 3 5 8 13 21 34
```

Python interpreter

```
>>> import fibo2
>>> fibo2.fib(100)
1 1 2 3 5 8 13 21 34
```

Modules

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

A package may be distributed with pip

- Arcade is an example of package

Packages

Package concept (II)

```
arcade/
├── __init__.py
├── __main__.py
├── sprites/
│   ├── __init__.py
│   ├── sprite.py
│   ├── animated.py
│   └── ...
├── types/
│   ├── __init__.py
│   ├── color.py
│   └── ...
├── color/
│   └── __init__.py
└── ...
```

(Arcade 3.3.3 source code)

Packages

Importing a package (I)

```
arcade/
├── __init__.py
├── __main__.py
├── sprites/
│   ├── __init__.py
│   ├── sprite.py
│   ├── animated.py
│   └── ...
├── types/
│   ├── __init__.py
│   ├── color.py
│   └── ...
├── color/
│   └── __init__.py
└── ...
```

Importing the whole package

```
import arcade
red = arcade.types.Color(255, 0, 0, 255)
```

Alternative way to import an individual module

```
from arcade.types import Color
red = Color(255, 0, 0, 255)
```

Give an alias to a namespace

```
import arcade.types as foo
red = foo.Color(255, 0, 0, 255)
```

Packages

Importing a package (II)

Imagine we run `from arcade.tilemap import *`

- In theory, it would import the whole package
- In practice, it would take too much time
- ... and mess the namespace

`__init__.py` may define a list named `__all__`

- It contains the names that will be exposed by that package
    - Functions, variables, modules and classes

arcade/tilemap/__init__.py

```
...
__all__ = ["TileMap", "load_tilemap"]
...
```

Modules

The `__main__.py` file

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

- Useful in command line mode and IDEs

Linux/PowerShell console

```
$ python3 -m mymodule
```

Try to run `python -m arcade`!

- (arcade/__main__.py source code)

Modules

Imports good practices

Imports style convention

1. Standard library
   Build-in Python modules
2. Third-party libraries
   Installed with pip
3. Local modules
   Modules you created

Good practices

- Avoid wildcards
  `from module import *`
- Alphabetic order within each subgroup

```python
"""
Example module with well-
    organized imports
"""

# Standard library
import json
import os

# Third-party libraries
import arcade
import numpy as np

# Local modules
from config import SETTINGS
from utils.helpers import
    process_data

# Rest of the code
def main():
    pass
```

Packages

Installing packages

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

- Very similar to `apt-get` in Linux or `brew` in MacOS
- Automatic dependencies management

pip usage (from OS terminal)

```
$ pip install SomePackage
```

```
$ pip install Pillow
```

In VS Code, you should use pip in the terminal

Packages

`requirements.txt`

A project may contain a `requirements.txt` file

- It recreates an environment
- Plain text file with dependences
- Can define fixed or minimum version

Supported by different tools

- ... including VS Code

requirements.txt

```
arcade==2.6.17
seaborn>=0.12
numpy
pandas
```

pip usage (from OS terminal)

```
$ pip install -r requirements.txt
```

Virtual environments

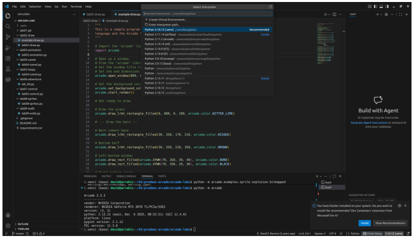
Versioning is problematic

- 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 and conda



Select Interpreter

Selected Interpreter: ./.venv/bin/python

```
+ Create Virtual Environment...
Enter interpreter path...
Python 3.10.12 (.venv) ./.venv/bin/python                        Recommended
Python 3.11.14 (airflow) ~/anaconda3/envs/airflow/bin/python      Conda
Python 3.11.2 (arcade) ~/anaconda3/envs/arcade/bin/python
Python 3.10.14 (ml) ~/anaconda3/envs/ml/bin/python
Python 3.10.9 (tf3) ~/anaconda3/envs/tf3/bin/python
Python 3.10.8 (tf) ~/anaconda3/envs/tf/bin/python
Python 3.9.18 (orange) ~/anaconda3/envs/orange/bin/python
Python 3.9.13 (base) ~/anaconda3/bin/python
Python ~/anaconda3/envs/tf2/python
Python ~/anaconda3/envs/yolo/python
Python 3.13.11 /bin/python3.13                                    Global
Python 3.13.11 /usr/bin/python3.13
Python 3.10.12 /bin/python3
Python 3.10.12 /usr/bin/python3
```

```python
"""
This is a sample program
language and the Arcade l
"""

# Import the "arcade" lib
import arcade

# Open up a window.
# From the "arcade" libra
# Set the window title to
# Set the and dimensions
arcade.open_window(800, 6

# Set the background colo
arcade.set_background_col
arcade.start_render()

# Get ready to draw

# Draw the grass
arcade.draw_lrbt_rectangle_filled(0, 800, 0, 200, arcade.color.BITTER_LIME)

# --- Draw the barn ---

# Barn cement base
arcade.draw_lrbt_rectangle_filled(30, 350, 170, 210, arcade.color.BISQUE)

# Bottom half
arcade.draw_lrbt_rectangle_filled(30, 350, 210, 350, arcade.color.BROWN)

# Left-bottom window
arcade.draw_rect_filled(arcade.XYWH(70, 260, 30, 40), arcade.color.BONE)
arcade.draw_rect_filled(arcade.XYWH(70, 260, 20, 30), arcade.color.BLACK)
```

```
(.venv) (base) david@arrakis:~/kk/pruebas-arcade/arcade-labs$ python -m arcade.examples.sprite_explosion_bitmapped
  warnings.warn(message, warning_type)
(.venv) (base) david@arrakis:~/kk/pruebas-arcade/arcade-labs$ python -m arcade

Arcade 3.3.3
------------
vendor: NVIDIA Corporation
renderer: NVIDIA GeForce RTX 3070 Ti/PCIe/SSE2
version: (3, 3)
python: 3.13.11 (main, Dec  6 2025, 08:52:51) [GCC 11.4.0]
platform: linux
pyglet version: 2.1.12
PIL version: 11.3.0
(.venv) (base) david@arrakis:~/kk/pruebas-arcade/arcade-labs$
```

Build with Agent

AI responses may be inaccurate. Generate Agent Instructions to onboard AI onto your codebase.

You have Docker installed on your system. Do you want to install the recommended 'Dev Containers' extension from Microsoft for it?

File Edit Selection View Go Run Terminal Help
EXPLORER
ARCADE-LABS
lab02-draw.py
example-draw.py
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash
Install
Show Recommendations
SUGGESTED ACTIONS
Ln 8, Col 1 Spaces: 4 UTF-8 LF Python Finish Setup 3.10.12 (.venv)

Great with `requirements.txt`!!!

Cool code examples

Example 1: Open a web browser

browser.py

```python
import webbrowser

url = input('Give me an URL: ')

webbrowser.open(url)
```

Cool code examples

Example 2: Create a thumbnail

thumbnail.py

```python
from PIL import Image

size = (128, 128)
saved = "africa.jpg"

im = Image.open("africa.tif")
im.thumbnail(size)
im.save(saved)
im.show()
```

(Source)

africa.jpg

Cool code examples

Example 3: List the contents of a directory

```
import os

for i in os.listdir("/home/david/"):
    print(i)
```

```
from pathlib import Path

for i in Path("/home/david").iterdir():
    print(i)
```

Cool code examples

Example 4: 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)
```

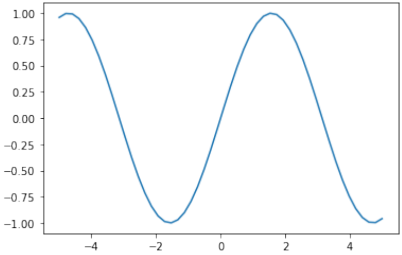
(Source)

Modules

Example 5: Plot

plot.py

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5)
plt.plot(x, np.sin(x))
```



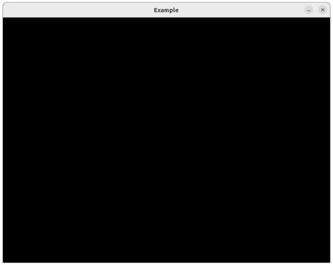
Modules

Example 6: Arcade

arcade.py

```
import arcade

WIDTH = 600
HEIGHT = 800

arcade.open_window(WIDTH, HEIGHT, "
    Example")

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 3.9+
- OpenGL capable hardware

Dependences

- Pyglet - Mutimedia library for Python

(Arcade web site)

Arcade

Open a Window (I)

Arcade has two APIs:

- Structured
- Object-Oriented

arcade.py - structured API

```python
import arcade

WIDTH = 600
HEIGHT = 800

arcade.open_window(WIDTH, HEIGHT, "Example")

arcade.run()
```

Arcade

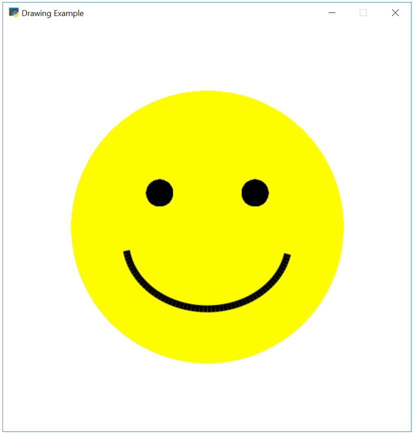
Drawing setup

drawing.py

```
import arcade

WIDTH = 600
HEIGHT = 800

arcade.open_window(WIDTH, HEIGHT,"Example")

arcade.set_background_color(arcade.color.WHITE)

arcade.start_render()

# Drawing here

arcade.finish_render()

arcade.run()
```

Arcade

Drawing (I)



(Source code)

Arcade

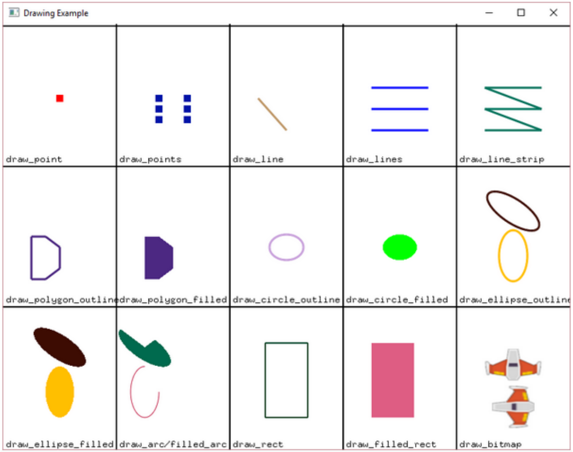
Drawing (II)

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_text()

draw_rect_filled()
draw_rect_outline()
draw_lrbt_rectangle_filled()
draw_lrbt_rectangle_outline()

draw_polygon_filled()
draw_polygon_outline()

load_texture()
draw_texture_rect()

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 API)

Arcade

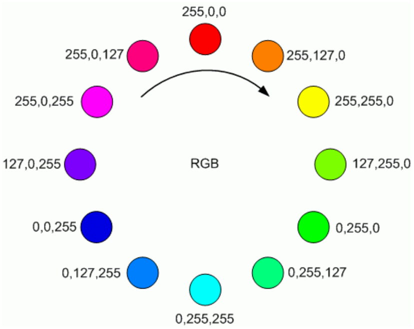
Colors

How can we know which colors has Arcade available?

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

Arcade

Colors: RGB



(Source code)