DEPARTMENT OF COMPUTER SCIENCE

University of Reading

# CS2PP: Programming in Python

# GUI Programming with PyQt

# Why I don't use Python

In the early 2000s, I invested heavily in learning **C**, **Java** and **Perl**:

- **C** — dominant systems/applications langauge
- **Java** — dominant client-side Web language
- **Perl** — major server-side Web language
- **PHP** — major server-side Web language, but mostly bad code written by beginners
- **Python** — toy language no-one took seriously

Whitespace sensitivity is a questionable design decision.

Some versions of Python changed the behaviour of operators between minor versions of the language.

# Why I don't use Python

In the 2020s:

- **C** — dominant systems language (but compare Rust?)
- **Java** — major applications language
- **Perl** — niche scripting language
- **PHP** — old-fashioned server-side Web language
- **Python** — dominant AI/data science language, but mostly bad code written by beginners

OK, I've written 1000s of lines of (bad beginner) Python code, but not 10,000s or 100,000s.

You can usually tell when an application is written in Python because it crashes frequently with a gigantic stack backtrace.

# Why I'm able to lecture Python

Application development is often more about learning to use particular libraries and their **API** (Application Programming Interface) than a particular programming language.

The most popular GUI libraries for Python are **Tk** and **Qt**.

Neither is written in Python, but there are Python **bindings** that let you call the library from within Python.

The difference between Qt GUI programming in Python and Qt GUI programming in another language is **mostly** syntactic... but there are some differences.

If I make elementary Python mistakes, like not knowing how to write a for loop... sorry.

If you have questions about the topics covered, feel free to ask, but I might have to get back to you.

# Hello World

# Hello World program

- `hello1.py` — first attempt at a GUI program
- `hello2.py` — adds all the important boilerplate that's easy to forget
- `hello3.py` — changing text size and alignment and window size

# Dice

# Dice program

- `dice1.py` — create a button
- `dice2.py` — make it do something with signals/slots
- `dice3.py` — make it do something with events
- `dice4.py` — more complex event behaviour

# Events, Handlers, Signals and Slots

GUI programming is usually event-driven.

The GUI toolkit provides code to draw **components** called **widgets** on the screen, and to read input (such as mouse movement, clicks and keypresses) from the user.

How does your program interact with this?

On modern operating systems, the desktop environment will run in a separate **process** or thread from applications. It keeps track of and draws the mouse pointer, as well as core GUI components such as windows.

Applications use a library or system calls to instruct the desktop environemnt to create components, to draw them, and to **poll** the desktop enviornment, asking if any **events** have occurred.

# Event Loops

In an imperative language like C, a GUI program consists of an **event loop** that looks like this:

```
setup();
while (e = nextEvent()) {
    switch (e.eventType) {
        case type1:
            ...
        case type2:
            ...
    }

}
```

When using a GUI toolkit library like Qt, typically the library implements the event loop and provides ways for you to register **callbacks** to your own code to handle events.

# Events and Event Handlers

**Events** encompass a wide variety of occurrences, including:

- mouse movement (press, release, move, drag, click, double-click)
- keyboard movement (press, release, type)
- GUI administration (window closed, moved, resized, exposed)
- timers
- in some libraries, other events such as disk/network I/O

In an object-oriented library, all the information about an event is encoded in an object.

A **handler** is a function that gets passed an event and takes action in response. In an object-oriented library, handlers are often written by subclassing a base component and overriding the default handler.

# Signals and Slots

Qt provides convenient high-level abstractions called **signals** and **slots**.

Roughly, **signals** are events generated by the GUI toolkit, while **slots** are **callback** functions.

Your program uses API calls to register a **slot** as a recipient of a **signal**.

Compared with the more traditional OOP approach to events and event handlers, this cuts down on the number of boilerplate classes you have to create.

It is usually better to use signals and slots, rather than events, where you can. But some unusual GUI behaviours might only be implementable using events.

# Dice program

- `dice5.py` — multiple GUI components
- `dice6.py` — text entry boxes
- `dice7.py` — vertical and horizontal layout
- `dice9.py` — input validation
- `dice10.py` — immediate response to input
- `dice11.py` — immediate response to input

# GUI toolkits

A GUI toolkit is a library that implements GUI components.

Some operating systems, such as Windows and MacOS, have integrated **native** GUI toolkits.

In the Linux world, the desktop environment is decoupled from the core operating system, so there are several toolkits available, including GTK, Qt, Motif, Tk, wxWidgets...

Most of these are also available on other operating systems, so you can build applications for Windows, MacOS or Linux without rewriting the GUI code.

Also, Wine has produced open source implementations of most Windows GUI components.

Java has several GUI tookits, including AWT and Swing.

# GUI toolkits — the native approach

How does a toolkit like GTK, Qt, AWT or Swing render widgets on Windows or MacOS?

One approach is to use the native widgets, so your application blends in with native applications.

But different toolkits support different widgets. Buttons, menus and windows are supported everywhere, but some specialist form controls are not.

This means a toolkit using this approach (such as original Java AWT) can't offer them.

Also, if you designed your application so it looked good on one OS, it might look weird with widgets from a different OS.

# GUI toolkits — full reimplementation approach

Another approach is to draw controls entirely within the toolkit's code, using the operating system's graphics primitives.

This means your application looks the same everywhere, but it blends in nowhere.

Also, it can be very slow.

Java's Swing toolkit used this approach, as did the original Qt.

# GUI toolkits — hybrid approach

There is a third way. The toolkit can use native widgets where available, but supplement them with custom widgets where no native equivalent exists.

This means applications blend in (as much as is possible) and are fast.

Modern Qt uses this approach.

It does make the library toolkit much more complex and harder to develop.

Sometimes it complicates distribution too, as a different binary is needed for each platform. This is no issue when using a compiled language like C++, but with supposedly cross-platform languages like Java, it can be an irritation.

# Dice program

- `dice12.py` — quit button with dialogue box
- `dice13.py` — menu bar with quit
- `dice14.py` — menu bar with roll and reset
- `dice15.py` — add a PNG image/logo

# Licensing: Qt

Qt was originally developed by TrollTech.

Licence was **open source**, but **"non-free"** (essentially not GPL-compatible).

If you made changes to the library source code, you couldn't redistribute them.

Suppose TrollTech had stopped making new versions open source. If the library was GPL or MIT licensed, open source developers could have forked it and continued to make updates for their own applications.

But as it wasn't, they would have been stuck using the old version, and it would have been a lot of work to migrate applications to a new toolkit.

# Licensing: Qt and KDE

When Linux desktop enviornments were being developed in the late 1990s, the emerging most popular enviornment, **KDE**, used Qt.

The community was sufficiently worried about the licence of Qt that many developers worked on the rival **GNOME** desktop, which used the GPL-licensed toolkit GTK.

The Qt licence was changed in the early 2000s, so this is no longer an issue.

But this is basically why GNOME is the "default" Linux desktop now, not KDE.

# Licensing: PyQt vs PySide

The developers of Qt focused on the native C++ API. Python wasn't a priority.

Riverbank Computing developed Python **bindings** called PyQt.

The PyQt bindings are released under the GPL. This means that if you use them in your program and distribute it, you must release source for your program under the GPL.

There is a less restrictive version of the GPL, called the LGPL ("L" is for "Lesser"). This allows distribution of programs that use LGPL libraries, without releasing the program under the GPL.

You can't distribute a closed-source commercial application that uses PyQt... unless you pay Riverside Computing for a commercial licence.

So the Qt Company, which currently maintaints Qt, developed competing bindings called PySide under a different licence.

# PyQt Documentation

Official Riverbank Computing documentation for PyQt, but much of it is incomplete or links to the Qt C++ documentation:

https://www.riverbankcomputing.com/static/Docs/PyQt5/

Official Qt and PySide documentation, but much of the Python documentation gives examples in C++:

https://doc.qt.io/

Excellent website with high-quality tutorials on PyQt, PySide, Tkinter:

https://www.pythonguis.com/

The author of the website has released reasonably priced e-books and paid-for video tutorials. If you are serious about PyQt development, buy them!

# Advanced Topics

**GUI Designer:** Qt has a graphical designer for GUIs. It can produce XML (which you can dynamically load in your code) or a Python module that you can include.

This makes it easier to build complex GUIs. You still need to plumb them in, though.

**Data Binding:** Suppose you want to display a table of data in a GUI control and allow editing. Rather than having to set each cell individually, you can **bind** a table GUI widget to a data structure, so it reads/writes it directly.

**Multithreading:** If you do something slow in an event handler, your GUI program will become unresponsive, as the event loop doesn't get to process events. You can fix this with multithreading.

**Graphics and Custom Widgets:** You can use the 2D graphics API to draw directly to your GUI, and even implement your own GUI widgets with custom rendering.

DEPARTMENT OF COMPUTER SCIENCE

University of Reading

# Next time: Web Programming with Flask