COMP120: Creative Computing

1: Tinkering in Python

Learning Outcomes

- **Outline** the role and basic functions of the IDE
- **Interpret** some basic Python code
- **Apply** pair programming practices to solve a simple text concatenation problem
- **Explain how** pictures are digitised into raster images by a computer system

FALMOUTH UNIVERSITY

Integrated Development Environment (IDE)

Using an IDE

- You *could* just write code in Notepad, but...
- An **Integrated Development Environment (IDE)** is an application providing several useful features for programmers, including:
  - A "run" button
  - Management of multi-file projects
  - Syntax highlighting
  - Autocompletion
  - Navigation
  - Language and API documentation
  - Debugging
  - Profiling
  - Version control

Setting up your own PC

- Python 3.6.7
  - `https://www.python.org/`
  - Python 2.x and Python 3.x are (slightly) different programming languages; we are using 3.x (for now)
  - Python is included with Mac OSX and most Linux distributions, but needs to be installed separately on Windows
- PyGame 1.9.6
  - We use `PyGame` as our framework for media computation and game development
  - Library version must accord with language version
  - Insteall on your PC using `pip`

```
pip install pygame==1.9.6
```

Setting up your own PC

- PyCharm 19.1.2
  - `https://www.jetbrains.com/student/`
  - Register with your `falmouth.ac.uk` email address to obtain PyCharm Professional Edition for free
  - Or, use the free open-source entitled ‘Community Edition’
  - Runs on Windows, Mac and Linux

PyCharm in the Lab

- You have to license your account to use PyCharm
- Run PyCharm and select **License server**
- In the **License server address** enter the following:

```
http://trlicefal.fal.ac.uk
```

- This will be added to your user profile and (hopefully) you will not need to do this again

Getting started with PyCharm

- Create a new project (from the start-up wizard or from the File menu)
- We want a “Pure Python” project
- Right-click the project in the panel on the left, and choose “New → Python File”
- Write some code!
- Setup the run configurations
- First run: click “Run → Run...” and choose the Python file
- Subsequent runs: click the ▶ button

Basic Python programs

Your first Python program

```
print("Hello, world!")
```

Your second Python program

```
print("This is a very long line of code which had to  ↩
    be split to fit on the slide, but you should type  ↩
    it as a single line.")
print("This is the second line of code.")
```

Assigning to variables

```
a = 10
print(a)
```

Variable	Value
a	

Remember!

- A program is a **sequence of instructions**
- The Python interpreter executes the **first line** of your program, then the **second line**, and so on
- When it reaches the end of the file, it **stops**

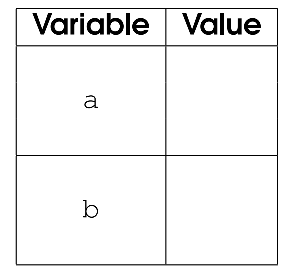
Socrative - FALCOMPMIKE

Login to Socrative!

`https://b.socrative.com/login/student/`

Reassigning variables (1)

```
a = 10
b = 20
b = a
print(a)
print(b)
```



Variable	Value
a	
b	

Reassigning variables (2)

```
a = 10
b = 20
a = b
print(a)
print(b)
```

Variable	Value
a	
b	

Reassigning variables (3)

```
big = 10
small = 20
big = small
print(big)
print(small)
```

Variable	Value
big	
small	

Reassigning variables (4)

```
a = 10
b = 20
a = b
b = a
print(a)
print(b)
```

Variable	Value
a	
b	

Reassigning variables (5)

```
a = 10
b = 20
c = 30

a = b
b = c

print(a)
print(b)
print(c)
```

Variable	Value
a	
b	
c	

Reading Input

```
print("Enter your name:")
name = input()

print("Enter your age:")
age = int(input())

print("Hello", name)
print("On your next birthday, you will be", age + 1, " ↩
    years old")
```

- `input()` reads a **string** (a sequence of characters—text) from the command line
- `int(...)` converts a **string** into an **integer** (a number)

Conditionals (1)

```
a = int(input())
b = 30

if a < 15:
    b = a

print(a)
print(b)
```

Variable	Value
a	
b	

Indentation

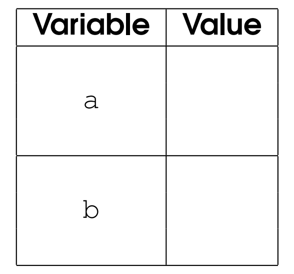
- Unlike many other programming languages, **indentation has meaning** in Python!
- Python uses indentation to denote the **block of code** inside a conditional, loop, function etc.
- PEP-8 recommends **4 spaces** for indentation
  - Some programmers use a tab character
  - **Never** mix tabs and spaces in the same file!
  - PyCharm inserts 4 spaces by default when you press the tab key; other IDEs and text editors can be configured to do this

Conditionals (2)

```
a = int(input())
b = 0

if a < 20:
    b = a + 1
elif a == 20:
    b = a * 2
else:
    a = 20
    b = 20

print(a)
print(b)
```



Variable	Value
a	
b	

Conditionals

An `if` statement can have:

- **Zero or more** `elif` clauses
- **An optional** `else` clause

In that order!

Mathematical operators

- + add
- - subtract
- * multiply
- / divide
- ** power

Order of operations: **BIDMAS**

- Brackets first
- Then indices (powers)
- Then division and multiplication (left to right)
- Then addition and subtraction (left to right)

Comparison operators

- `<` less than
- `<=` less than or equal to
- `>` greater than
- `>=` greater than or equal to
- `==` equal to
- `!=` not equal to

Note the difference between `=` and `==`

- `a = b` means “make `a` be equal to `b`”
- `a == b` means “is `a` equal to `b`?”

For loops and ranges

```
for i in range(5):
    print(i)
```

- `range(n)` is the **sequence** $0, 1, 2, \ldots, n - 1$
- So `range(5)` is the **sequence** $0, 1, 2, 3, 4$
- Note: `range(n)` **does not include** n
- The `for` loop iterates through the items in a sequence **in order**

For loops (1)

```
a = 0
b = 0

for i in range(5):
    a = i
    b = b + i

print(a)
print(b)
```

Variable	Value
a	
b	
i	

For loops (2)

```
a = 0
b = 0

for i in range(10):
    if (i < 3) or (i > ↩
        7):
        a += i
    else:
        b += i

print(a)
print(b)
```

Variable	Value
a	
b	
i	

While loops

The `while` loop keeps executing while the condition is **true**

```
a = 1

while a < 100:
    a = a * 2

print(a)
```

Variable	Value
a	

Looping forever

```
a = 1

while True:
    a = a * 2
    print(a)
```

Summary

We have seen some basic code constructions in Python

- `print()` and `input()` for command-line input and output
- Variable assignment using `=`
- `if` statements for choosing whether or not to execute a block of code
- `for` loops to execute a block of code a specified number of times
- `while` loops to execute a block of code until a condition is no longer true

These are enough to write some simple programs, but you will see several more in coming weeks...

Challenge

- In pairs
- **Implement** the code excerpt
- **Fix** the errors in the code excerpt
- **Modify** the code excerpt to incorporate functions and arguments
- **Post** your solution to the `#comp120` slack channel

You can learn more about functions and arguments at:

`https://docs.python.org/3/tutorial/controlflow.html#defining-functions`

Challenge

The function:

```
def madlib()
```

Should become:

```
def madlib(name, pet, verb, snack)
```

Challenge

```
def madlib():
    name = 'Link'
    pet = 'Spyro'
    verb = 'ate'
    snack = 'doughnuts'
    line1 = 'once upon a time,' + name + ' walked'
    line2 = ' with ' + pet + ', a trained dragon.'
    line3 = 'Suddenly, ' + pet + ' announced,'
    line4 = 'I really want some ' + snack + '!'
    line5 = name + ' complained. Where am I going to  ↩
        get that?'
    line6 = 'Then ' + name + 'found a wizard's wand.'
    line 7 = 'With a wave of the wand, '
    line8 = pet + ' got ' + snack + '. '
    line9 = 'Perhaps surprisingly, ' + pet + ' ' + ↩
        verb + '  ' + snack
    print line1 + line2 + line3 + line4
    print line5 + line6 + line7 + line8 + line9
```

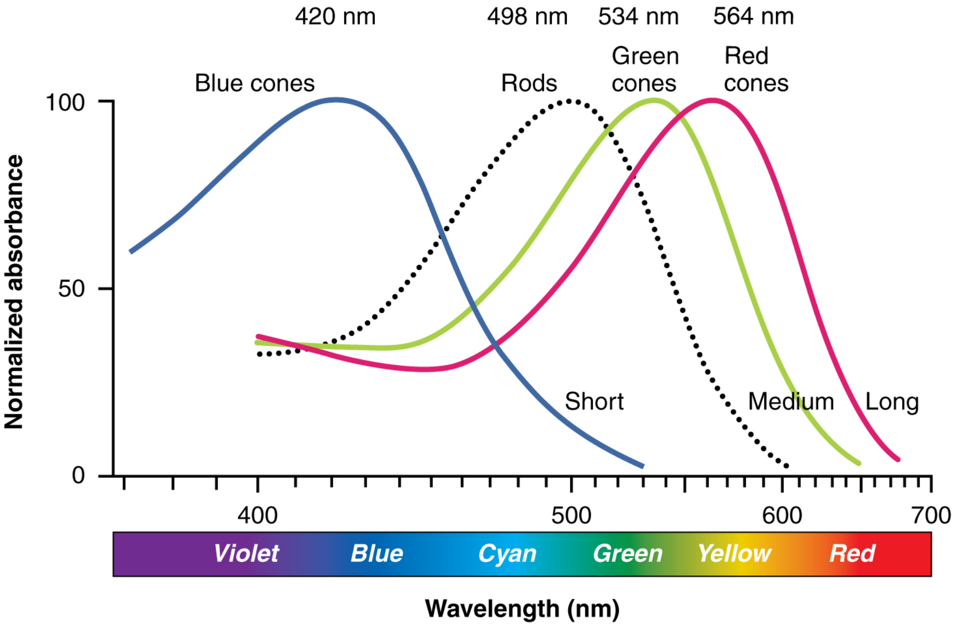
Tinkering Graphics

Light Perception

- Colour is continuous:
  - Visible light is in the wavelengths between 370nm and 730nm
  - i.e., 0.00000037 — 0.00000073 meters
- However, we *perceive* light around three particular peaks:
  - Blue peaks around 425nm
  - Green peaks around 550nm
  - Red peaks around 560nm

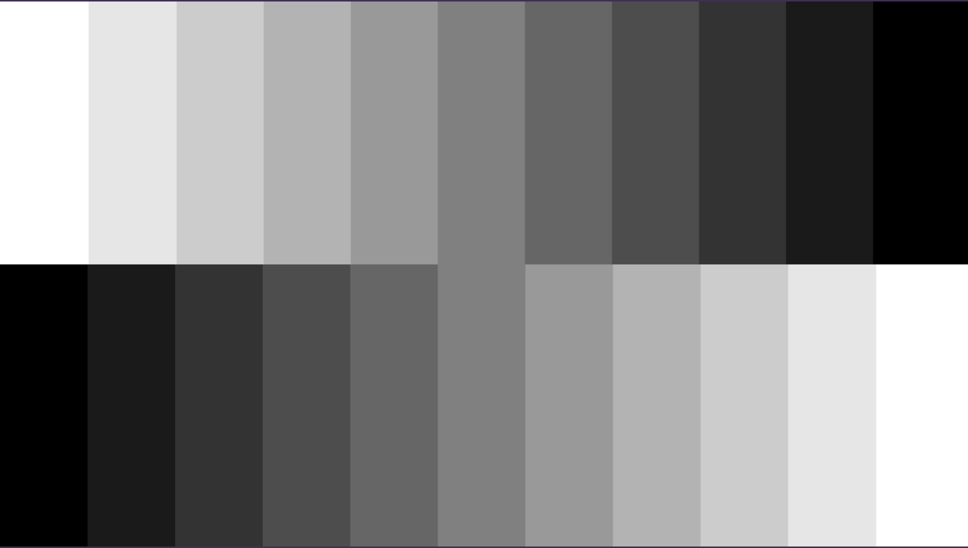
Light Perception

- Our eyes have three types of colour-sensitive photoreceptor cells called ‘cones’ that respond to light wavelengths
- Our perception of colour is based on how much of each kind of sensor is responding
- An implication of this is perception overlap: we see two kinds of ‘orange’ — one that’s spectral and one that’s combinatorial


420 nm
498 nm
534 nm
564 nm
Blue cones
Rods
Green cones
Red cones
100
50
0
Normalized absorbance
Short
Medium
Long
400
500
600
700
Violet
Blue
Cyan
Green
Yellow
Red
Wavelength (nm)

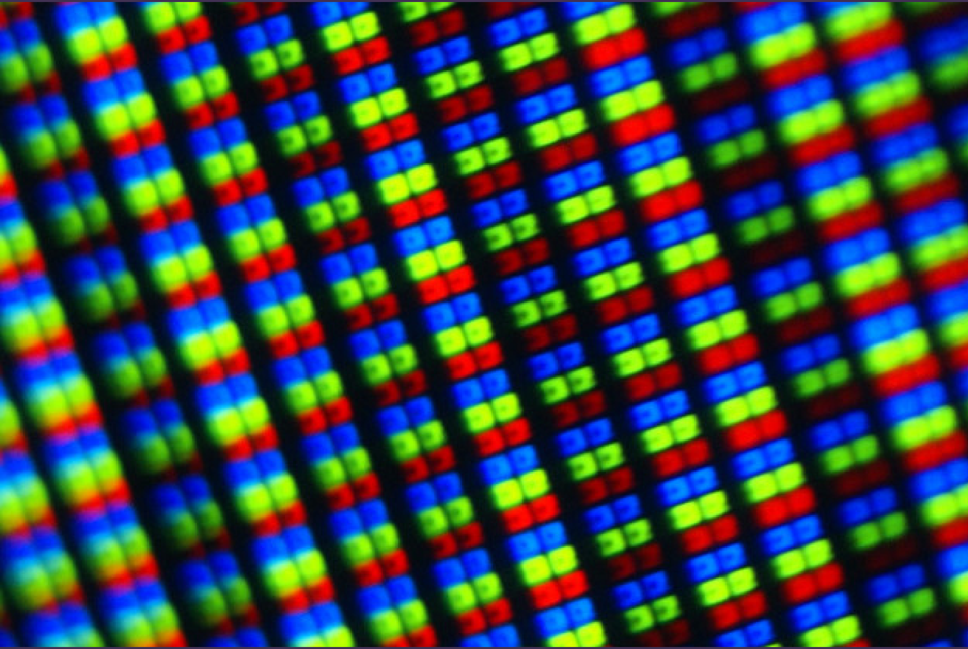
Luminance vs Colour

- Our eyes have another type of photoreceptor cells called ‘rods’ that respond to light intensity
- Our perception, however, is actually luminance: a relativistic contrast of *borders* of things (i.e., motion)
  - Luminance is *not* the amount of light, but our perception of the amount of light
  - Much of our luminance perception is based on comparison to background, not raw values
- An implication of this is perception overlap: we see blue as ‘darker’ than red when the intensity is actually the same



Resolution

- We have a limited number of rods and cones in our eyes
- This means humans perceive vision in a limited resolution — yet, we perceive vision as continuous
- We take advantage of this human characteristic in computer monitors



Pixels

- We digitize pictures into many little dots
- Enough dots and it looks like a continuous whole to our eye
- Each element is referred to as a *pixel*

Pixels

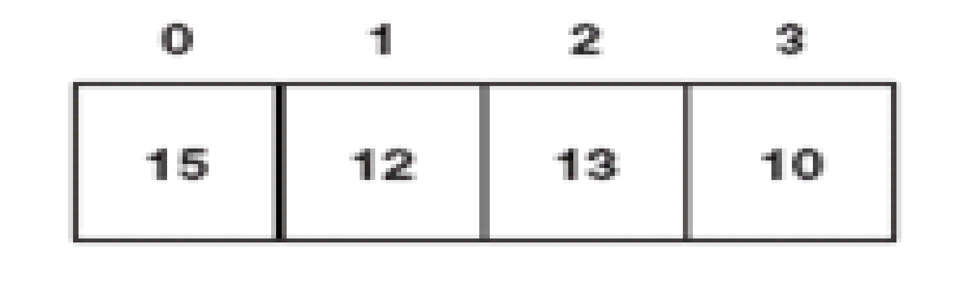
Pixels must have:

- a color
- a position

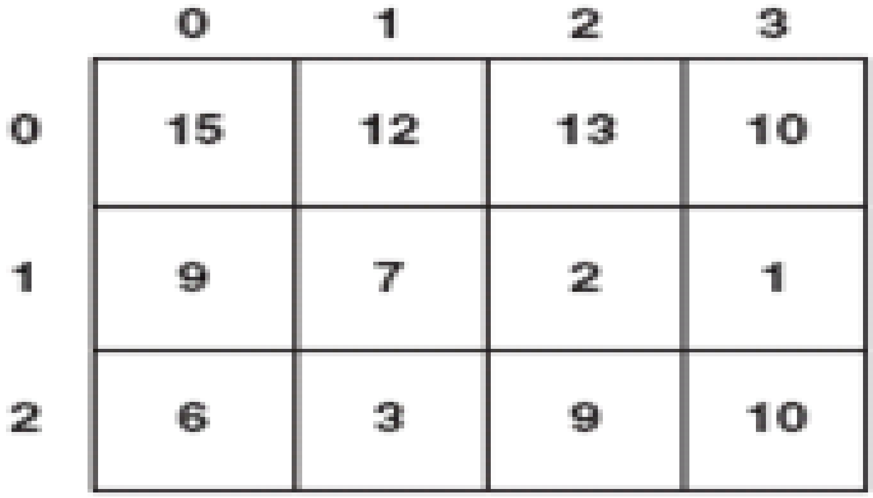
Pictures and Surfaces

In PyGame, a `Surface` is a *matrix* of pixels

- It is not a continuous line of elements, that is, a one-dimensional *array*
- A picture has two dimensions: width and height
- It's a two-dimensional *array*



0	1	2	3
15	12	13	10



	0	1	2	3
0	15	12	13	10
1	9	7	2	1
2	6	3	9	10

Pictures and Surfaces

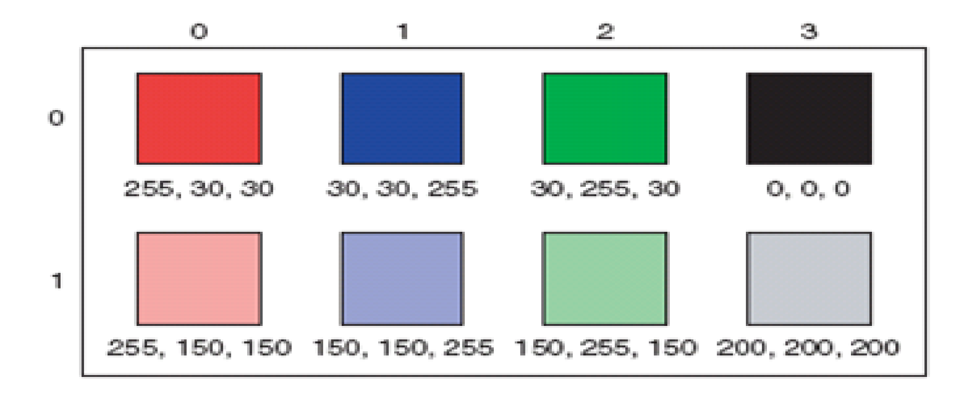
- (x, y) —or— (horizontal, vertical)
- The origin (0,0) is top-left
- (1,0) = 12
- (0, 2) = 6

Encoding Colour

- Each element in the matrix is a pixel, with the matrix defining its position and the value defining its colour
- Computer memory stores numbers, so colour must be encoded into a number:
  - CMYK = cyan, magenta, yellow, black
  - HSB = hue, saturation, brightness
  - RGBA = red, green, blue, alpha (transparency)
- By default, PyGame uses RGBA

Encoding RGB

- Each component color (red, green, and blue) is encoded as a single byte
- Colors go from `(0,0,0)` to `(255,255,255)`:
  - If all three components are the same, the colour is in grey-scale
  - `(0,0,0)` is black
  - `(255, 255, 255)` is white



	0	1	2	3
0	🟥 255, 30, 30	🟦 30, 30, 255	🟩 30, 255, 30	⬛ 0, 0, 0
1	🟥 255, 150, 150	🟦 150, 150, 255	🟩 150, 255, 150	⬜ 200, 200, 200

Encoding Bits

Why `255`?

- If we have one bit, we can represent **TWO** patterns:
  - 0
  - 1
- If we have two bits, we can represent **FOUR** patterns:
  - 00
  - 01
  - 10
  - 11
- With n bits, we can have 2^n patterns
- With 8 bits, there will be 256 patterns
- One of these patterns will be 0, so the highest value we can represent with 8 bits is: $2^8 - 1$, or 255

Encoding Bits

- RGB uses 24-bit color (i.e., $3 * 8 = 24$)
  - That's 16,777,216 possible colours
  - Our eyes cannot discern many colours beyond this
  - A challenge is display technology: monitors and projectors can`t reliably reproduce 16 million colours
- RGBA uses 32-bit colour
  - No additional colour, but offers support for transparency
- Assuming 1 `byte` == 8 `bits`
- We can use this information to estimate the size of a bitmap:
  - 320x240x24 = 230,400 bytes
  - 640x480x32 = 1,228,800 bytes
  - 1024x768x32 = 3,145,728 bytes