CSCI 145 · Quiz 9 Reference Sheet (Lectures 17–18)

IMAGES ARE GRIDS OF PIXELS

	0	1	2	3
0	(0, 0)	(1, 0)	(2, 0)	(3, 0)
1	(0, 1)	(1, 1)	(2, 1)	**(3, 1)**
2	(0, 2)	(1, 2)	(2, 2)	(3, 2)

Width 4, height 3. The shaded pixel is `img[3][1]`

- `img[x][y]`: column x, row y. **x first**, as in (x, y)
- `(0, 0)` is the top-left corner
- x grows to the right, y grows **down**
- x goes from 0 to `img.width - 1`, y from 0 to `img.height - 1`

GRAYSCALE AND COLOR

- **Grayscale** (mode `"L"`): each pixel is one number, its brightness, from 0 (black) to 255 (white). One byte per pixel
- **Color** (mode `"RGB"`): each pixel is a tuple `(red, green, blue)`, each part 0 to 255. Three bytes per pixel

(255, 0, 0)	red	**(255, 255, 0)**	yellow
(0, 255, 0)	green	**(255, 255, 255)**	white
(0, 0, 255)	blue	**(0, 0, 0)**	black

- **Bit depth**: the bits per pixel. k bits give 2^k grays (8 bits: 256, 4 bits: 16, 2 bits: 4, 1 bit: black or white)
- **Resolution**: an image has width × height pixels

MIDDIMAGE

`from midd_media import middimage`	
`middimage.open(filename)`	The image in a file
`middimage.new(width, height)`	New black color image
`middimage.new(width, height, "L")`	New black gray image
`img.width, img.height`	Columns, rows; no `()`
`img.mode`	`'L'` or `'RGB'`
`img[x][y]`	The pixel at (x, y)
`img[x][y] = value`	Sets that pixel
`img.show()`	Opens it in a window
`img.save(filename)`	Saves it as a file
`img.copy()`	A new copy of `img`

PIXEL VALUES

- Values must be ints. A float is a `TypeError`: use `int()` on a calculation
- A value above 255 is stored as 255; below 0 is stored as 0
- An x or y outside the image is an `IndexError`
- Unpack a color pixel: `r, g, b = img[x][y]`
- Or index it: `img[x][y][0]` is its red part
- Set a color pixel to a whole tuple: `img[x][y] = (r, g, b)`

EVERY PIXEL: NESTED LOOPS

```
def lighten(img, factor):
    result = middimage.new(img.width, img.height, "L")
    for x in range(img.width):
        for y in range(img.height):
            result[x][y] = int(img[x][y] * factor)
    return result
```

- For each x, the inner loop runs through every y: column by column, width × height pixels in all
- The pattern: make a new image, visit every pixel, set each new pixel from the original, return the new image. `img` doesn't change
- A function can change the image it is given instead; then it returns nothing
- Gray value of a color pixel: `int((r + g + b) / 3)`

OBJECTS AND CLASSES

- An **object** bundles data, its **attributes**, with functions for that data, its **methods**
- A **class** is a blueprint for a kind of object. Each object made from it is an **instance**, with its own attributes
- Class names are in CamelCase: `Card`, `MiddImage`

`obj.attribute`	Reads an attribute; no `()`
`obj.attribute = value`	Changes it
`obj.method(arguments)`	Calls a method on `obj`

WRITING A CLASS

```
RANKS = ["2", "3", "4", "5", "6", "7", "8",
         "9", "10", "J", "Q", "K", "A"]

class Card:
    """A playing card"""
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit
    def __str__(self):
        return self.rank + " of " + self.suit
    def value(self):
        return RANKS.index(self.rank) + 2
    def beats(self, other):
        return self.value() > other.value()
```

- `__init__`, the **constructor**, runs when a new object is made and sets up its attributes. Two underscores on each side
- Every method's first parameter is `self`, the object itself. Python passes it for us
- `self.rank = rank` gives the object an attribute. Inside the class, always write `self.` to use an attribute or call a method
- `__str__` **returns** a string; it doesn't print. `print` and `str` call it for us. Use `str()` to join a number to a string
- In `c1.beats(c2)`, `self` is `c1` and `other` is `c2`

USING A CLASS

`c1 = Card("Q", "spades")`	A new Card: 2 arguments
`c1.rank`	`'Q'`
`c1.value()`	12
`c1.beats(c2)`	True
`print(c1)`	Prints `Q of spades`

VARIABLES NAME OBJECTS

- `c2 = c1` makes a second name for the **same** object. A change made through one name shows through the other
- `Card(...)` always makes a **new** object, even with the same arguments
- A method can change its object: `self.plays += 1`

OBJECTS THAT HOLD OBJECTS

- An attribute can be any value, even a list of objects: a `Deck` has `self.cards`, a list of `Card` objects that starts as `[]`
- A method can loop over them: `for card in self.cards:`
- `Deck()`: `__init__` has only `self`, so we pass nothing

COMMON ERRORS

No `self` parameter: `def value():`	`TypeError`
No `self.` in `__init__`: `rank = rank`	`AttributeError`
No `self.` in a method: `RANKS.index(rank)`	`NameError`
Wrong number of arguments: `Card("Q")`	`TypeError`
`__str__` prints instead of returning	`TypeError`
No parentheses: `c1.value`	No call