CSCI 145 · Quiz 10 Reference Sheet (Lectures 19–20)

SPECIAL METHODS

You write	Python calls	You write	Python calls
`print(a)`	`a.__str__()`	`a + b`	`a.__add__(b)`
`a == b`	`a.__eq__(b)`	`a * b`	`a.__mul__(b)`
`a < b`	`a.__lt__(b)`		

```
class Rational:
    def __init__(self, num, den):
        g = math.gcd(num, den)
        self.num = num // g
        self.den = den // g
    def __str__(self):
        return str(self.num) + "/" + str(self.den)
    def __eq__(self, other):
        return self.num == other.num and self.den == other.den
    def __add__(self, other):
        return Rational(
            self.num * other.den + other.num * self.den,
            self.den * other.den)
```

- Without `__eq__`, `==` asks "same object?", just like `is`
- `self` is the left side, `other` the right. `__add__` returns a **new** object

ENCAPSULATION

- An object bundles its data with the methods that work on it. Code outside the class calls the methods, not the attributes
- A leading underscore, as in `_data`, means "private". Python doesn't stop you: it is a signal to other programmers

INHERITANCE

```
class Pet:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def birthday(self):
        self.age = self.age + 1
    def speak(self):
        return "..."
    def __str__(self):
        return self.name + " says " + self.speak()

class Dog(Pet):
    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed
    def speak(self):
        return "Woof"

class Cat(Pet):
    def speak(self):
        return "Meow"
```

- `class Dog(Pet):` makes Dog a **child class** (subclass) of its **parent class** (superclass) Pet. A Dog "is a" Pet
- A child class **inherits** every method of its parent. It can add attributes and methods, and **override** a method: same name, new body
- `super().__init__(name, age)` runs Pet's `__init__`: no `self`
- Cat has no `__init__`, so `Cat("Mochi", 2)` uses Pet's

WHICH METHOD RUNS? POLYMORPHISM

- Python looks in the object's own class first, then in the parent class
- That is the **method resolution order**. When an inherited method calls `self.speak()`, the search starts again in the object's own class
- **Polymorphism**: the same call works on many kinds of objects, and each runs its own class's version: `pet.speak()` is `'Woof'` for a Dog, `'Meow'` for a Cat

COMMON ERRORS

No `super().__init__(...)` in Dog's `__init__`	`AttributeError`
Passing `self`: `super().__init__(self, name, age)`	`TypeError`
Too many arguments: `Cat("Mochi", 2, "tabby")`	`TypeError`
A misspelled override: `def speek(self):`	No override

THE GAME LOOP

```
pygame.init()                              # always first
screen = pygame.display.set_mode((600, 400))
clock = pygame.time.Clock()
running = True
while running:
    for event in pygame.event.get():       # events
        if event.type == pygame.QUIT:
            running = False
    ball.update()                          # update
    screen.fill((255, 255, 255))           # draw
    ball.draw(screen)
    pygame.display.flip()                  # show
    clock.tick(60)                         # and wait
pygame.quit()                              # always last
```

- One trip around the loop makes one **frame**. Closing the window is a `QUIT` event, and `running = False` ends the loop
- `fill` paints over the last frame. `flip()` puts the finished frame on the screen. `tick(60)` waits: at most 60 frames a second

THE SCREEN AND DRAWING

- `(0, 0)` is the upper-left corner. x grows to the right, y grows **down**. A negative y is above the screen
- A color is `(red, green, blue)`, each from 0 to 255

```
pygame.draw.circle(screen, color, (x, y), radius)
pygame.draw.rect(screen, color, (x, y, width, height))
```

- A circle is placed by its **center**. Its edges are `radius` away: the left edge is at `x - radius`, the top at `y - radius`
- A rectangle is placed by its **upper-left corner**. Later shapes are drawn on top of earlier ones

A BALL: MOVE, THEN CHECK THE WALLS

Move	`self.x = self.x + self.vx`
Past the left or right wall?	`if self.x - self.radius < 0 or self.x + self.radius > SCREEN_WIDTH:`
Then reverse	`self.vx = -self.vx`
Top and bottom	The same, with y, vy, and `SCREEN_HEIGHT`

PYGAME.RECT AND THE PLAYER

`pygame.Rect(x, y, width, height)`	Corner, then size
`r.x, r.y, r.width, r.height`	Change x, y to move it
`a.colliderect(b)`	`True` if they overlap

- A player has one attribute, `rect`, a `pygame.Rect` that is `PLAYER_SIZE` wide and tall. `player.rect.x` is its left edge
- `player.move(dx, dy)` keeps it on the screen: `min` stops it at the right or bottom edge, `max` stops it at 0

```
self.rect.x = max(0, min(self.rect.x + dx,
                         SCREEN_WIDTH - PLAYER_SIZE))
```

- Held keys, asked every frame: `keys = pygame.key.get_pressed()`, then `if keys[pygame.K_LEFT]:` (also `K_RIGHT`, `K_UP`, `K_DOWN`). Up is a negative `dy`. A `KEYDOWN` event happens once per press

ROCKS AND COLLISIONS

- Attributes x, y (the center), `radius`, and `speed`

`rock.reset()`	Moves it to a new random spot above the top
`rock.update()`	Moves down by `speed`. Then, if `self.y - self.radius > SCREEN_HEIGHT` (the whole rock is off the bottom), calls `self.reset()`
`rock.get_rect()`	Returns the square around the rock, a new `pygame.Rect(x - radius, y - radius, 2 * radius, 2 * radius)`
A hit?	`player.rect.colliderect(rock.get_rect())`

- `game_over` is a **flag**, a variable that is `True` or `False`. After a hit nothing moves, but the loop keeps drawing