CSCI 145 · Exam 4 Reference Sheet (Lectures 15–20) — page 1 of 2

BITS AND BINARY (BASE 2) — Lecture 15

- n bits give 2^n patterns: the unsigned numbers 0 to $2^n - 1$
- Nibble = 4 bits, byte = 8 bits (0 to 255), word = 64 bits on most laptops

n	0	1	2	3	4	5	6	7	8	9	10
2^n	1	2	4	8	16	32	64	128	256	512	1024

Place	128	64	32	16	8	4	2	1

- Binary to decimal: add the places that hold a 1
- Decimal to binary: subtract the biggest place that fits, write 1 there, repeat. Or divide by 2: `num % 2` is the last bit, `num // 2` drops it; read the remainders bottom up
- **Adding:** 0 + 1 = 1, 1 + 1 = 10 (write 0, carry 1), 1 + 1 + 1 = 11 (write 1, carry 1). The sum can need one more bit

HEXADECIMAL (BASE 16) AND COLORS — Lecture 15

- Digits 0–9, then A–F for 10–15. One hex digit = 4 bits; two hex digits = one byte, `00` to `FF` = 0 to 255
- Hex to decimal: first digit × 16 + second digit (see the × 16 column)
- Decimal to hex: `num // 16` is the first digit, `num % 16` the second
- Binary ↔ hex: four bits at a time, from the right
- Color `#RRGGBB`: red, green, blue, one byte each; 2^{24} colors

Dec	Binary	Hex	× 16
0	0000	0	0
1	0001	1	16
2	0010	2	32
3	0011	3	48
4	0100	4	64
5	0101	5	80
6	0110	6	96
7	0111	7	112

Dec	Binary	Hex	× 16
8	1000	8	128
9	1001	9	144
10	1010	A	160
11	1011	B	176
12	1100	C	192
13	1101	D	208
14	1110	E	224
15	1111	F	240

`0b1010`	A binary int literal (10)
`0x1a`	A hex int literal (26); either case
`bin(num)`	A **string** starting `'0b'`, no leading zeros
`hex(num)`	A **string** starting `'0x'`, lowercase digits
`int(text, 2)`	A binary string to an int
`int(text, 16)`	A hex string to an int; either case

NEGATIVE NUMBERS: TWO'S COMPLEMENT — Lecture 16

- **Unsigned**: every place is positive; 8 bits hold 0 to 255
- **Signed two's complement**: the leftmost place is negative; the rest are as before. Every negative number starts with 1

8 bits	−128	64	32	16	8	4	2	1

- **Negate** a number: flip every bit, then add 1. The same two steps work in both directions
- Add as before; drop a carry out of the left end
- **Overflow**: the answer needs more bits than there are
- n bits hold -2^{n-1} to $2^{n-1} - 1$: 4 bits −8 to 7, 8 bits −128 to 127

FLOATS ARE APPROXIMATE — Lecture 16

- Ints are exact; floats are not. Many values, like 0.1, can't be stored exactly: there are gaps between floats
- Sums of halves, quarters, eighths, … are exact
- Round before you show a float: `round(x, 2)`
- Don't compare floats with ==; check that they're close: `abs(a - b) < 0.000001`
- To stop at zero, test `f <= 0`, not `f == 0`
- A 64-bit float can be far bigger than 2^{64}; big floats have big gaps

TEXT AND ASCII — Lecture 16

- Each character is stored as a number, its **code**. Strings compare by code
- Uppercase and lowercase letters are 32 apart; letters and digits are in order
- ASCII: 128 codes (7 bits). Unicode: every language plus emoji; its first 128 codes match ASCII

`ord(ch)`	The code of one character (an int)
`chr(code)`	The character with that code (a string)

	×16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	32	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

Row = first hex digit, column = second. Decimal code = the row's × 16 + the column (A = 10, …, F = 15). sp = space

IMAGES ARE GRIDS OF PIXELS — Lecture 17

	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**
- `(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** (mode `"L"`): each pixel is one number 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)` is red, `(255, 255, 0)` yellow, `(255, 255, 255)` white, `(0, 0, 0)` black
- **Bit depth**: the bits per pixel; k bits give 2^k grays. **Resolution**: width × height pixels

MIDDIMAGE — Lecture 17

`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, img.mode`	Attributes: no `()`
`img[x][y]`	The pixel at (x, y)
`img[x][y] = value`	Sets that pixel
`img.show(), img.save(filename)`	Show it, save it
`img.copy()`	A new copy of `img`

- Pixel 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 it to a whole tuple: `img[x][y] = (r, g, b)`

EVERY PIXEL: NESTED LOOPS — Lecture 17

```
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
- Gray value of a color pixel: `int((r + g + b) / 3)`

CSCI 145 · Exam 4 Reference Sheet (Lectures 15–20) — page 2 of 2

OBJECTS AND CLASSES — Lecture 18

- An **object** bundles data, its **attributes**, with its **methods**. A **class** is a blueprint; each object made from it is an **instance**, with its own attributes

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

class 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: `c1 = Card("Q", "spades")` passes 2 arguments
- `c1.rank` is an attribute: no `()`. `c1.value()` calls a method. In `c1.beats(c2)`, `self` is `c1` and `other` is `c2`
- Every method's first parameter is `self`, the object itself. Python passes it for us. 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
- `c2 = c1` makes a second name for the **same** object. A change made through one name, or by a method (`self.plays += 1`), shows through the other. `Card(...)` always makes a **new** object
- An attribute can be a list of objects: `self.cards = []` in a `Deck`. A method can loop over it: `for card in self.cards:`

SPECIAL METHODS AND ENCAPSULATION — Lecture 19

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:        # a fraction: attributes num, den
    def __eq__(self, other):
        return self.num == other.num and self.den == other.den
    def __add__(self, other):
        num = self.num * other.den + other.num * self.den
        return Rational(num, 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**: code outside the class calls the methods, not the attributes. A leading underscore, as in `_data`, signals "private"

INHERITANCE AND POLYMORPHISM — Lecture 19

```
class Pet:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    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 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?** Python looks in the object's own class first, then in the parent class, even when an inherited method calls `self.speak()`
- **Polymorphism**: the same call works on many kinds of objects; each runs its own class's version: `pet.speak()` is `'Woof'` for a Dog

COMMON ERRORS — Lectures 18–19

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 `super().__init__(...)` in Dog's `__init__`	`AttributeError`
Passing `self`: `super().__init__(self, name, age)`	`TypeError`
A misspelled override: `def speek(self):`	No override

THE GAME LOOP — Lecture 20

```
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
- Held keys, asked every frame: `keys = pygame.key.get_pressed()`, then `if keys[pygame.K_LEFT]:`

THE SCREEN AND DRAWING — Lecture 20

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

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

BALL, PLAYER, AND ROCK — Lecture 20

```
Ball.update(self)           # x, y: the center
    self.x = self.x + self.vx
    if (self.x - self.radius < 0
            or self.x + self.radius > SCREEN_WIDTH):
        self.vx = -self.vx
    # the same for y, vy, and SCREEN_HEIGHT

Player.move(self, dx, dy)   # stays on the screen
    self.rect.x = max(0, min(self.rect.x + dx,
                             SCREEN_WIDTH - PLAYER_SIZE))
    # the same for y, dy, and SCREEN_HEIGHT

Rock.update(self)           # x, y: the center
    self.y = self.y + self.speed
    if self.y - self.radius > SCREEN_HEIGHT:
        self.reset()    # a new random spot above the top

Rock.get_rect(self)         # the square around the rock
    return pygame.Rect(self.x - self.radius,
                       self.y - self.radius,
                       2 * self.radius, 2 * self.radius)
```

- `pygame.Rect(x, y, width, height)`: corner, then size. `player.rect` is one, `PLAYER_SIZE` wide and tall; `player.rect.x` is its left edge
- `a.colliderect(b)` is `True` if two Rects overlap. A hit: `player.rect.colliderect(rock.get_rect())`
- Up is a negative `dy`. `game_over` is a **flag**, a variable that is `True` or `False`. After a hit nothing moves, but the loop keeps drawing