

# Warmup (L4)

---

- Write a function

`checkWithin(result, expected, tolerance, name)`

that generalizes our **assert**-based tolerance checking

- (That is, it **asserts** that `result` is within the tolerance of `expected`, with `name` as the name/message for the assertion)
- Note: Don't use this in your submissions; MarkUs won't detect the separate tests 😊

# CS114

---

## Module 2: Making decisions

# Making decisions

---

CS114 M2

# Assertions

---

- Remember “==” and “<” from our assertions?
- What do they actually do?

```
print(hypotenuse(4, 5) < 7)  
True
```

# Conditionals

---

- `hypotenuse(4, 5) < 7` is simply a fact: it is true
- We can *conditionalize code* based on facts
  - That is, rather than just asserting that something is true as a test, we check if it's true, then choose what to do next

# Why?

---

- Remember:  
computation =  
calculation +  
repetition +  
*decision making*
- We did calculation in Module 1
- Conditions will give us *decision making*

# Decision making

```
def pos(x: float) -> float:
```

```
    """
```

```
    If x is not zero, return the absolute value of x.  
    Otherwise, return 1.
```

```
    """
```

```
    if x < 0:
```

```
        return -x
```

```
    elif x == 0:
```

```
        return 1
```

```
    else:
```

```
        return x
```

```
assert pos(42) == 42,
```

```
assert pos(-42) == 42,
```

```
assert pos(0) == 1, "Special case pos(0) is 1."
```

This code only runs if  $x < 0$

This code only runs if  $x == 0$   
(`elif` means "else if")

This code only runs if  $x > 0$   
That condition is implicit: it  
only runs if neither previous  
case matched.

# Decision making

---

```
def pos(x: float) -> float:
```

```
    """
```

```
    If x is not zero, return the absolute value of x.  
    Otherwise, return 1.
```

```
    """
```

```
    if x < 0:
```

```
        return -x
```

```
    elif x == 0:
```

```
        return 1
```

```
    else:
```

```
        return x
```

Indenting again to show what happens conditionally and what doesn't

```
assert pos(42) == 42, "Absolute value of positive is positive."
```

```
assert pos(-42) == 42, "Absolute value of negative is positive."
```

```
assert pos(0) == 1, "Special case pos(0) is 1."
```

# Decision making

---

```
def multiOp(x: float, y: float, op: int) -> float:
    """
    Return "x op y", where the operation is given as
    a code in op:
        0: addition
        1: subtraction
        2: multiplication
    """

    assert op >= 0, "No negative operation codes"
    assert op <= 2, "Operation codes go up to 4"

    if op == 0:
        return x + y
    if op == 1:
        return x - y
    return x * y

assert multiOp(1, 1, 0) == 2, "Simple addition"
assert multiOp(4, -3, 2) == -12, "Simple multiplication"
# ... more tests ...
```

# Decision making

---

```
def multiOp(x: float, y: float, op: int) -> float:
    """
    Return "x op y", where the operation is given as
    a code in op:
        0: addition
        1: subtraction
        2: multiplication
    """
    assert op >= 0, "No negative operation codes"
    assert op <= 2, "Operation codes go up to 4"
    if op == 0:
        return x + y
    if op == 1:
        return x - y
    return x * y

assert multiOp(1, 1, 0) == 2, "Simple addition"
assert multiOp(4, -3, 2) == -12, "Simple multiplication"
# ... more tests ...
```



We write  $\leq$  for  $\leq$  and  $\geq$  for  $\geq$

# Decision making

---

```
def multiOp(x: float, y: float, op: int) -> float:
    """
    Return "x op y", where the operation is given as
    a code in op:
        0: addition
        1: subtraction
        2: multiplication
    """
    assert op >= 0, "No negative operation codes"
    assert op <= 2, "Operation codes go up to 4"
    if op == 0:
        return x + y
    if op == 1:
        return x - y
    return x * y

assert multiOp(1, 1, 0) == 2, "Simple addition"
assert multiOp(4, -3, 2) == -12, "Simple multiplication"
# ... more tests ...
```

else/elif are not required



# Still imperative

---

- Anything indented under `if` is executed conditionally
- Anything after `if` (unindented) is executed *unconditionally*. It's simply run in order.
  - Except for *early return*
- Let's add some `prints` to our `multiOp` function in Jupyter to understand what is and isn't run

# Decision making

---

```
def multiOp(x: float, y: float, op: int) -> float:
    """
    Return "x op y", where the operation is given as
    a code in op:
        0: addition
        1: subtraction
        2: multiplication
    """

    assert op >= 0, "No negative operation codes"
    assert op <= 2, "Operation codes go up to 4"

    if op == 0:
        return x + y
    if op == 1:
        return x - y
    return x * y

assert multiOp(1, 1, 0) == 2, "Simple addition"
assert multiOp(4, -3, 2) == -12, "Simple multiplication"
# ... more tests ...
```

# Variable naming aside

---

- I told you to use descriptive names, but I just named my parameters “x” and “y”
- Names are for what the variable means *to the function*, not to whoever calls the function
- In the case of `multiOp`, `x` means nothing to us, so `x` is as good as any other name
- `op` gets a real name because it means something to `multiOp`!

# In-lecture quiz (L4)

---

- <https://student.cs.uwaterloo.ca/~cs114/quiz/>
- Q1: Which of these Python function definitions is valid?

A. `def return(x: int) -> int:  
 return x`

B. `def roundBadly(x: float) -> int:  
 if x < 0:  
 return int(x) - 1  
 return int(x)`

C. `def greater(x: float, y: float) -> float:  
 if x > y:  
 return x  
 return y`

D. `void iGotLost(const std::string &user) {  
 cout << "Aren't you glad you're learning "  
 "Python instead of C++?"  
 << std::endl;  
}`

# In-lecture quiz (L4)

---

- <https://student.cs.uwaterloo.ca/~cs114/quiz/>

- Q2: What will this code print?

```
def f(x: int) -> None:
    if x > 5:
        print("Big")
    print("Small")
f(42)
```

- A. Big
- B. Small
- C. Big  
Small
- D. Nothing

# Boolean logic

---

CS114 M2

# Booleans

---

- `True` and `False` are *values*
- Although it's weird, you can, e.g., store it in a variable:

```
x = hypotenuse(4, 5) < 7
```

- Just as `True` is a value, so is `False`:

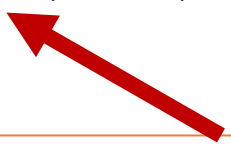
```
print(hypotenuse(4, 5) > 7)  
False
```

# Booleans

---

- These are called *boolean values*, named for logician George Boole
  - For the type checker, `"bool"`
- Math with booleans is called *boolean logic*
- We get booleans with our comparators:

`==, !=, <, <=, >, >=`



`!=` for  $\neq$  (not equal to)

# Multiple conditions

---

- You can put `ifs` in `ifs`

```
if op < 2:  
    if op == 0:  
        return x + y  
    else:  
        return x - y  
else:  
    return x * y
```

- Let's add some `prints` to understand this

# Multiple conditions

---

- There are also operators to combine conditions:

```
if op >= 0 and op <= 2:  
    # ... do it ...  
else: # op out of range  
    return -1
```

- **and** for both, **or** for either

# Multiple conditions

---

- There are also operators to combine conditions:

```
if op >= 0 and op <  
    # ... do it ...  
else: # op out of r  
    return -1
```

It's sometimes useful to comment an **else** to make it clear when it happens.

- **and** for both, **or** for either

# Nesting vs. combining

---

- When you put an `if` inside of another `if`, that's called *nesting* conditionals
- Sometimes it's unavoidable (or would be ugly to avoid)
  - In particular, when you need to nest a condition *and* do something else
- When you can avoid it, you usually should. It results in *pyramids of doom* (code that gets indented so far that it's annoying to read)
  - But use common sense!

# A note on **or**

---

- In common use, “**or**” can be ambiguous
  - If CS114 is my favorite class *or* I fail it, I’ll remember it well.
  - What if CS114 is your favorite class *and* you fail it?
- In CS, “**or**” always means “and/or”, so, e.g., “ $1 == 1$  **or**  $2 == 2$ ” is true.

# Complex combos

---

- You can also invert a condition with **not**, and group things with parentheses just like in numerical math

```
if not (op < 0 or op > 2) :  
    # ... do it ...  
else: # op out of range  
    return -1
```

# Mind your precedence

---

- BEDMAS is now BEDMASCN&O  
(pronounced bed-masc-nando)
- Brackets/parentheses, exponents, division and multiplication, addition and subtraction, ...
- Conditionals (`==`, `!=`, `<`, `<=`, `>`, `>=`)
- **not**
- **and**
- **or**

# Mind your precedence

---

- Confused?
- When in doubt, just use parentheses to make it clear
- Don't double up (e.g., `((a<b))` ).  
Otherwise, it's never bad style.
  - (Using parentheses around **assert** isn't bad style, it's just incorrect.)

# Booleans are *values*

---

- Here's a new version of `multiOp`:

```
def multiOp(x: float, y: float, op: int) -> float:
    """
    ...
    """
    if opInRange(op):
        if op == 0:
            return x + y
        elif op == 1:
            return x - y
        return x * y
    else:
        return -1
```

- Let's write `opInRange` to work with it.

# Booleans are *values*

---

- Here's a new version of `multiOp`:

```
def multiOp(x: float, y: float, op: int) -> float:
```

```
    """
```

```
    ...
    """
```

```
    if opInRange(op):
```

```
        if op == 0:
```

```
            return x
```

```
        elif op == 1:
```

```
            return x
```

```
        return x * y
```

```
    else:
```

```
        return -1
```

There's no comparison here (at least, not directly!). We got a `bool` because that's what `opInRange` returned!

- Let's write `opInRange` to work with it.

# Booleans are *values*

---

- Here's a new version of `multiOp`:

```
def multiOp(x: float, y: float, op: int) -> float:
```

```
    """
```

```
    ...
    """
```

```
    if opInRange(op):
```

```
        if op == 0:
```

```
            return x
```

```
        elif op == 1:
```

```
            return x
```

```
        return x ^ y
```

```
    else:
```

```
        return -1
```

Making the `op == 2` condition totally implicit is usually poor style, because it's unclear. I did it here just to show an **elif** without an **else**.

- Let's write `opInRange` to work with it.

# The power of abstraction

---

CS114 M2

# Nonobvious conditions

---

- Let's write a function `isEven` to check if an integer is even.
- None of our comparators look like "is even" or "divisible by"...
- New operator! `%`
  - Remainder after division, e.g., `5 % 2 == 1`
  - Called "modulo"

# Modulo and quotient

---

- In math, remainder after division is usually paired with *quotient* to keep division in integers
- We can do the same to keep division in `ints`.
- Quotient is `//` (two slashes)

# Modulo

---

- Wait, remainder after division still isn't "is even" or "divisible by"...
- A number is even if it's divisible by 2...
- A number is divisible by  $y$  if the remainder after division by  $y$  is 0...
- So, we can use  $==$ :  $x \% y == 0$

# isEven

---

- With modulo in mind, let's write our `isEven` function.

# isEven

---

- With modulo in mind, let's write our `isEven` function.

```
def isEven(v: int) -> bool:
    """
    Returns True if v is even,
    False otherwise.
    """
    return v % 2 == 0
```