COMP 1510 Winter 2022

Programming Methods

BCIT

Week 10: Syntactic sugar, sets, and functions 2.0

1

Agenda for week 10

1. Syntactic sugar and list and dictionary comprehensions
2. Nested data structures
3. Pass statement
4. Syntactic sugar and conditional expressions
5. Sets
6. More about unit testing
    1. Generating input for tests
    2. Testing printed output
    3. Creating 'predictable' random numbers
7. More about functions:
    1. Default values
    2. Variable length parameter lists
    3. Positional and arbitrary arguments
    4. Keyword arguments
    5. Annotations
8. Building good functions:
    1. Encapsulation
    2. Information hiding
    3. Message passing
    4. Decomposition
    5. Testing)

9 March 2022 COMP 1510 202210 2

2

LIST COMPREHENSION

3

Syntactic sugar

- Syntax in a language designed to make things easier to read and express
- It makes the language “sweeter” for humans
- Things can be expressed
  - More clearly
  - More precisely
  - In an alternative, preferred style.
- When syntactic sugar is removed from a language, the functionality and expressive power are not diminished

4

List comprehensions

- This is an example of syntactic sugar
- An example is worth a thousand words, so check this out!

```
>>> squares = [value ** 2 for value in range(1, 11)]
>>> squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```


List comprehension how-to

1. Begin with a descriptive name of the **list**
2. Open a set of **square brackets**
3. Define the **expression** for the values you want in the list
4. Write a **for loop** to generate the numbers you want to feed into the expressions
5. Close the **square brackets.**

Compare, and savour the sweetness:

```
new_list = []
for i in old_list:
    if filter(i) == True:
        new_list.append(expression(i))
```

```
new_list = [expression(i) for i in old_list if filter(i)]
```


List comprehensions

- We can use a list comprehension to create a new list containing only the elements you don't want to remove:

```
my_list = [x for x in orig_list if not determine(x)]
```

- What does this do?

```
listOfWords = ["this","is","a","list","of","words"]
items = [ word[0] for word in listOfWords ]
print(items)
```

More examples

```
>>> [x.lower() for x in ["A","B","C"]]
['a', 'b', 'c’]
>>> [x.upper() for x in ["a","b","c"]]
['A', 'B', 'C’]

>>> string = "Hello 12345 World”
>>> numbers = [x for x in string if x.isdigit()]
>>> print(numbers)
['1', '2', '3', '4', '5’]
>>> letters = [x for x in string if x.isalpha()]
```


More examples

```
def double(x):
    return x * 2

 >>> [double(x) for x in range(10)]
 >>> [double(x) for x in range(10) if x % 2 == 0]
```

Aside: we can loop through a slice

This is useful if we want to loop through a subset of the elements in a list:

```
>>> squares = [value ** 2 for value in range(1, 11)]
>>> for value in squares[:3]:
        print(value) # prints the first three only
```

Remember we can copy a list using a slice of the whole list: list_name[:]

DICTIONARY COMPREHENSION

The dictionary comprehension

Similar to a list comprehension:

- Group the expression using curly braces instead of square brackets
- Left part of the for keyword expresses both a key and a value, separated by a colon

```
>>> letters = {i : chr(65 + i) for i in range(4)}
>>> letters
{0:'A', 1:'B', 2:'C', 3:'D'}
```

9 March 2022 COMP 1510 202210 13

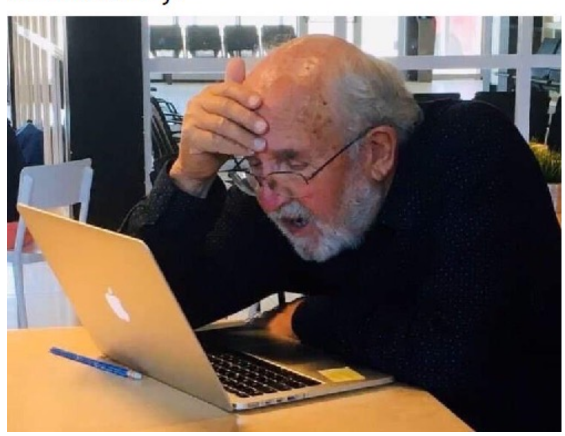
13

Why do we need this?

- Sometimes we have data arranged in a sequence of length-2 tuples or equivalent
- Provides a succinct idiom that replaces the traditional loop (just like the list comprehension)
- *Improves code clarity*

Me on Friday: I'll stop here and pick up where I left off on Monday

Me on Monday:



9 March 2022 COMP 1510 202210 14

14

What can we do with it?

```
list_of_email_addrs =
    ["Chris@bcit.ca", "Billy@microsoft.org", "Guido@python.org"]
contacts = {address.lower() : 1 for address in list_of_email_addrs}
print(contacts)
{‘chris@bcit.ca' : 1, ‘billy@microsoft.org' : 1, 'guido@python.org' : 1}

sums  = {(k, v): k + v for k in range(4) for v in range(4)}
print(sums)
{(0, 0): 0, (0, 1): 1, (0, 2): 2, (0, 3): 3, (1, 0): 1, (1, 1): 2, (1, 2):
3, (1, 3): 4, (2, 0): 2, (2, 1): 3, (2, 2): 4, (2, 3): 5, (3, 0): 3, (3,
1): 4, (3, 2): 5, (3, 3): 6}
```


15

What else?

```
def invert(d):
    return {v : k for k, v in d.items()}

letters = {0 : 'A', 1 : 'B', 2 : 'C', 3 : 'D’}
inverted_letters = invert(letters)
print(inverted_letters)

{'A' : 0, 'B' : 1, 'C' : 2, 'D' : 3}
```


16

Which is sweeter?

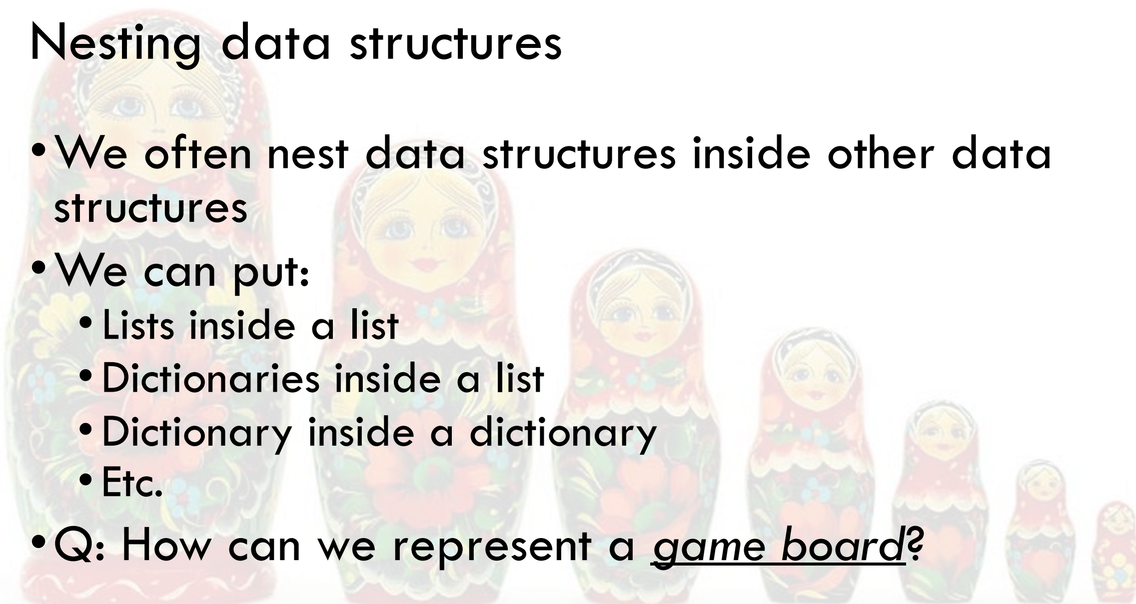
```
counts = []
for _ in range(26):
    counts.append(0)
letters = [chr(letter) for letter in range(65, 91)]

tally = dict(zip(letters, counts))
print(tally)

tally_2 = { letter: count for letter in letters for count in counts}
print(tally_2)
```


NESTED DATA STRUCTURES

Nesting data structures



- We often nest data structures inside other data structures
- We can put:
  - Lists inside a list
  - Dictionaries inside a list
  - Dictionary inside a dictionary
  - Etc.
- Q: How can we represent a *game board*?

Lists of dictionaries

- We can make a simple dictionary that represents a character in a game
- What if we had a party (group) of characters?
- We probably want to make a list!

```
cleric = create_character('Alvin')
ranger = create_character('Margaret')
mage = create_character('Drucilla')
party_of_adventures = [cleric, ranger, mage]
```

More examples:

1. A dictionary in a dictionary:
   - Suppose we have several users for a website, each with a unique name
   - We can use usernames as keys in a dictionary, and the value will be a dictionary that stores information about each user.
2. A list in a dictionary of meals:
   - A pizza being served for lunch has a list of toppings
3. A dictionary of tuples:
   - Associate keys (city names) with tuples (geographic coordinates)

What gets printed?

```
for row in range(5):
    for column in range (5):
        print(“Hello”)


for row in range(5):
    for column in range (5):
        print(row + column)


for row in range(5):
    for column in range (5):
        print("Row" + str(row), "Column" + str(column))
```

What gets printed?

```
for row in range(5):
    for column in range (5):
        print("(" + str(row) + ", " +
              str(column) + ")", end=' ')
    print()
```

Check out this sample code: multiplication_table.py

Can you fix the doctest?

What gets printed?

```
grades_keanu = [98, 90, 43, 87, 65]
student_summary = {"keanu": grades_ keanu}

grades_setareh = [65, 87, 48, 85, 98]
student_summary["setareh"] = grades_ setareh

for student, grades in student_summary.items():
    for grade in grades:
        if grade > 95:
            print(student + " earned an A+!")
```

PASS STATEMENT

What if we need a placeholder?

- Sometimes we need to:
  - Define a function
  - Create an incomplete if statement or loop
- We want to create syntactically correct structure, but we don't have any commands to execute
- We can use the **pass** statement
- It's just a placeholder. It does nothing. Nada. Nic. 아무것도. Rien. Niente. **没有**. Юу ч биш. **何もない**. कुछ भी तो नहीं. هیچ چیزی.Ничего такого.

Pass examples

```
while True:
    pass

def function_name(parameters):
    """I don't know what this does yet.  But the
    assignment specification says I need it. I
    will define this placeholder idea later…"""
    pass
```


CONDITIONAL EXPRESSIONS

Syntactic sugar in Python

- The list comprehension is a good example
- The dictionary comprehension is another good example
- We can remove it
- We can still do everything without it
- But the code we write without it is just not as “Pythonic”

- Another common piece of syntactic sugar in programming languages is the **conditional expression**

Conditional expression

- Sometimes called the *ternary operator*:

```
X if C else Y
```

1. First evaluates C
2. If C is True, then X is evaluated, and the value of X is returned
3. Otherwise, if C is False, Y is evaluated, and the value of Y is returned

```
>>> age = 12
>>> status = 'minor' if age < 21 else 'adult'
>>> status
'minor'
```

SETS

31

We've looked at some neat containers:

- **string** (a sequence of Unicode elements)
- **list** (a mutable sequence of anything)
- **tuple** (an immutable sequence of anything)
- **dictionary** (a sequence of key-value pairs)

There is another collection you should know about:

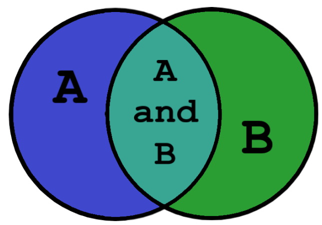
the set

32

What's so special about a set anyway?

- Lists:
  - Maintains sequence
  - Can contain duplicates
  - Mutable
- Tuples
  - Maintains sequence
  - Can contain duplicates
  - Immutable
- Dictionaries
  - Does not maintain sequence (it's unordered)
  - Cannot contain duplicate keys
  - Can contain duplicates values
  - Mutable

- Set:
  - Does **not** maintain sequence (it's unordered)
  - **Cannot contain duplicates**
  - Mutable



Python set

```
>>> vowels = { "A", "E", "I", "O", "U"}
>>> vowels
{'U', 'E', 'O', 'I', 'A'}
```

- It looks like a list of elements , except it uses curly braces { }
- Don't confuse a set with a dictionary which also uses curly braces { }
- A dictionary contains comma separated pairs of things related to one another with the colon :
- Python optimizes storage to speed retrieval in a set, so items are not guaranteed to stay in insertion order.

The set *cannot* contain duplicates

This is a mathematical rule that informs programming, i.e., we do what math tells us:

```
>>> my_list = [2, 3, 2, 5]
>>> my_set = set(my_list)
>>> my_set
{2, 3, 5}
```

Of course there's a function for this!

Be mindful of this, though:

```
>>> { 'a', 'e', 'i'} == { 'a', 'e', 'i', 'a'}
True
```


The set is a type, just like list, integer, etc.

```
>>> vowels = { 'a', 'e', 'i', 'o', 'u'}
>>> type(vowels)
<class 'set'>
```

Well, almost the same. There's a special way to create an empty set:

```
>>> a = set( )
>>> a
set( )
```

*We cannot use { } because that creates an empty dictionary!

Some set operations

- In mathematics, set operations include:
  - Adding an element
  - Removing an element
  - Emptying a set
- In Python we can use these familiar-looking methods:
  - add(element) to add element to the set
  - remove(element) to remove element from the set
  - clear( ) to remove all elements from the set

Why use a set

- Suppose we are recording observations of birds around Lost Lagoon
- Suppose all volunteer observations are aggregated in a simple hand-written list like this:
  1. Mallard
  2. Mallard
  3. Mud hen
  4. Great blue heron
  5. Wood duck
  6. American coot aka mud hen aka pouldeau
- We can use a set to remove the duplicates!

Some important notes

- <u>Set contents must be immutable</u>
- Checking for set membership is lightning fast
- It uses a mathematical technique called hashing (see COMP 2522)
- Hashing only works when the values are immutable
- Mutable values like lists cannot be added to sets because mutable values are not hashable

```
>>> s = set( )
>>> l = [1, 2, 3]
>>> s.add(l) # Throws a TypeError!!
```

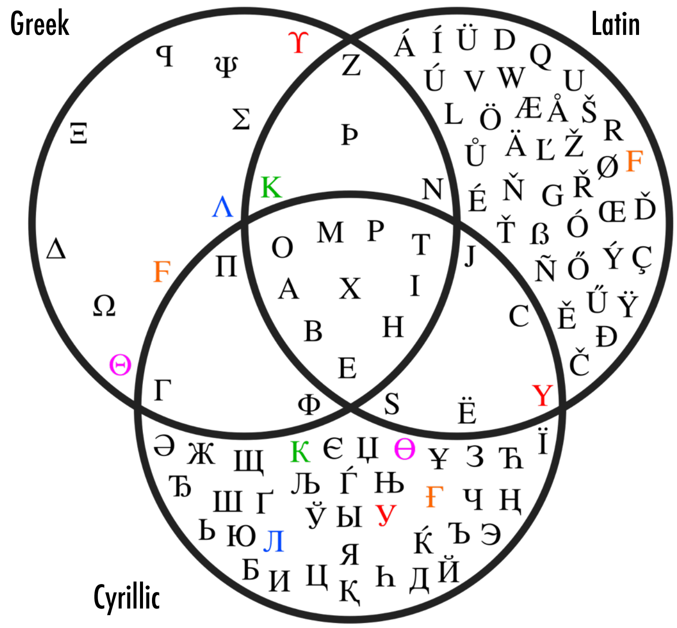

More set operations

Suppose I have a set called s and a set called other:

1. s.difference(other) will return a set of items that occur in set s but not in set other
2. s.intersection(other) will return a set of items that occur in set s and in set other
3. s.union(other) will return a set of items that occur in set s and/or in set other
4. s.symmetric_difference(other) will return a set of items that are in either s or other but not in both sets
5. s.issubset(other) returns true iff s is a subset of other
6. s.issuperset(other) returns true iff other is a subset of s

Can you find…

1. greek.difference(cyrillic)
2. cyrillic.intersection(latin)
3. latin.union(greek)
4. latin.symmetric_difference(greek)
5. (cyrillic.union(greek)).difference(latin)
6. (cyrillic.difference(latin)).difference(latin)
7. (greek.intersection(latin)).union(cyrillic)



41

MORE ABOUT UNIT TESTS

42

What is the purpose of the unit test?

1. *Test* functions that cannot be adequately tested with doctests
2. *Demonstrate* that postconditions are true when a function is correctly invoked by a user who meets the preconditions
3. *Generate* mock objects that represent externalities so we can control what our function receives from external helper functions
4. *Redirect* standard output and test what functions print
5. *Create* predictable random numbers
6. *Assert* that an expected error correctly occurs when required in response to specific data.

9 March 2022 COMP 1510 202210 43

43

Each unit test must:

1. Assemble
2. Act
3. Assert

9 March 2022 COMP 1510 202210 44

44

What should we add to test our code?

- A collection of unit tests is a collection of test functions inside a class (sort of like a module, we will learn about classes later this term!)
- Each test is a function that contains one **assertion** and in rare cases two assertions
- All the assertions from all the tests we write for a function must test all the **disjointed equivalency partitions**
- *The unit test functions must all begin with the word test*

Which assertions can we make?

https://docs.python.org/3/library/unittest.html#assert-methods

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Fixtures and mocking

- Sometimes a function requires help
- When a function is totally independent, it is easy to test
- But many functions aren't independent
- Many functions use other functions as aids when performing their tasks
- The other functions can be built-in functions, or functions from other libraries, or functions from elsewhere in the same module!
- In order to test a function that uses all these helpers, we must control what the helpers do
- We create mock (fake) versions that the unit testing framework uses when our function executes and asks for help!

Generating "random" input for tests

- When we annotate or "decorate" a unit test function with a @patch annotation right before it, we are creating a mock object
- A mock object is a fake object (remember functions are objects too!)
- We use it to control the value(s) our tested function receives from the helpers it invokes
- We can do this with functions like random.randint( ):

```
@patch('random.randint', return_value=5)
@patch('random.randint', side_effect=[5, 3, 1])
```

Check out test_simple_game.py

Generating user input for tests

- What if we have a function that accepts user input?
- Let me remind you how we can patch user input!
- This is amazing! Cue fireworks and a ticker-tape parade! 🎉🎊🎉
- We can patch the input function
- Remember that the input function is a built-in function
- It belongs to the builtins namespace:

```
@patch('builtins.input', side_effect=["north", "quit"])
```

Check out test_simple_game.py

Testing printed output

- What if we have a function that prints to the screen
- How can we "capture" that output to make sure it's right?
- We can mock standard output
- Create a mock object that represents standard output
- Patch something called sys.stdout
- We can redirect output to a mutable string object called StringIO from the io module like this:

```
@patch('sys.stdout', new_callable=io.StringIO)
```

Check out test_simple_game.py

FUNCTIONS 2.0

51

The Python function’s formal definition

```
Funcdef ::=
[decorators] "def" funcname "(" [parameter_list] ")"
             ["->" expression] ":" suite
```

Where anything in square brackets is optional:

1. Annotations (decorators) like @patch
2. Parameter lists can be empty
3. -> expression is something we will look at in a few slides.

52

We've been using **positional** arguments

- When we create a function that requires input, we use meaningful names for the parameters
- To use (invoke) the function, we must pass arguments to it
- **The arguments to the function must be provided in a specific order**
- Values matched this way are called *positional arguments*.

We can also use **keyword** arguments

- We've seen a few examples of this (the list's sort method, the random module's choices function)
- A keyword argument is a **name-value pair we pass to a function**
- We **DON'T** have to provide input in the correct order
- We **DO** have to use the correct function parameter name
- **Benefit**: makes our code much **easier to read**
- **Benefit**: wrong order won't break the code.

Keyword argument example

```
def validate(list, validator):
    """Does stuff.  Maybe it goes through the list
    and makes a new one that contains elements
    approved by the validator. It's not important."""

def main():
    """Drive the program."""
    log = list(…)
    min = some_value
    result = validate(validator=min, list=log)
    result = validate(list=log, validator=min) # same!
    result = validate(log, min) # same!
```

9 March 2022 COMP 1510 202210 55

55

Default values

- We can add default values to function parameters
- <u>This makes parameters optional</u> when invoking the function
- **Caveat:** parameters with default values need to be listed after all the parameters that don't have default values
- Why? So the interpreter can correctly interpret positional arguments

```
def format_name(first, last, middle=None):
    if middle:
        # do something
    else:
        # do something else
…
formatted_name = format_name('Neil', 'Harris', 'Patrick')
formatted_name = format_name('Cher', 'Bono')
```

9 March 2022 COMP 1510 202210 56

56

Avoid using mutable default values

- Default parameter values are evaluated from left to right when the function definition is executed
- The default parameter is evaluated once, when the function is defined, and the same "pre-computed" value is used for each call (the function stores the address of the value)
- This is important to understand when a default parameter is a mutable object, such as a list or a dictionary
- **If the function modifies the mutable object (e.g. by appending an item to a list), the default value is in effect modified and stays modified until the function is called again, at which time it might be modified yet again!**
- This is bad!

Solution: use None to avoid this

Readme: https://docs.python-guide.org/writing/gotchas/

- Use None as the default
- Explicitly test for none in the function
- Great example from the python.org pages:

```
def add_pizza_sauce(toppings=None):
    if toppings is None:
        toppings = []
    toppings.append("sauce")
    return toppings
```

Variable length parameter lists

- Sometimes we don't know how many arguments a function needs to accept (it's an *arbitrary or unpredicatable* amount)
- We can use a variable length example list:

```
def make_pizza(*toppings)
    """Makes a pizza with the toppings."""
```

- The asterisk in the parameter name *toppings means something special to the interpreter
- Python will create a tuple called toppings that contains all of the arguments passed to the function when it is called
- A tuple is created even if only one value is passed

Variable length parameter list example

```
def make_pizza(*toppings):
    """Makes a pizza."""
    for topping in toppings:
        # add it to the pizza and do stuff.

…

make_pizza( ) # makes a plain pizza
make_pizza('green peppers') # inoffensive I guess
make_pizza('ham', 'pineapple', 'garlic') # nasty
```

Mixing positional and arbitrary args

- If a function will accept several kinds of arguments, the parameter that accepts an arbitrary number of arguments *must be placed last*
- Python matches positional and keyword arguments first
- Python collects any remaining arguments in the final parameter tuple

```
def make_pizza(size, *toppings) 👍
def make_pizza(*toppings, size) ❌
```


Using arbitrary keyword arguments

- Q: What if we want to accept an arbitrary number of arguments, but **we don't know what kind of information** will be passed to the function?
- A: Accept as many key-value pairs as possible
- We do this using a ***double asterisk***
- Python will create a dictionary and pack the key-value pairs it receives into this dictionary
- Inside the function we treat it as a dictionary

Using arbitrary keyword arguments

```
def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know about a user."""
    profile = {}
    profile['first_name'] = first
    profile['last_name'] = last
    for key, value in user_info.items():
        profile[key] = value
    return profile
…
user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')
print(user_profile)
```


I also ❤️ function annotations!

- Described in PEP (Python Enhancement Proposal) 3107
- Completely optional
- Used for parameters and return values
- Enhance code readability
- Compare:

```
def add(a, b):
def add(a: int, b: int) -> int:
```

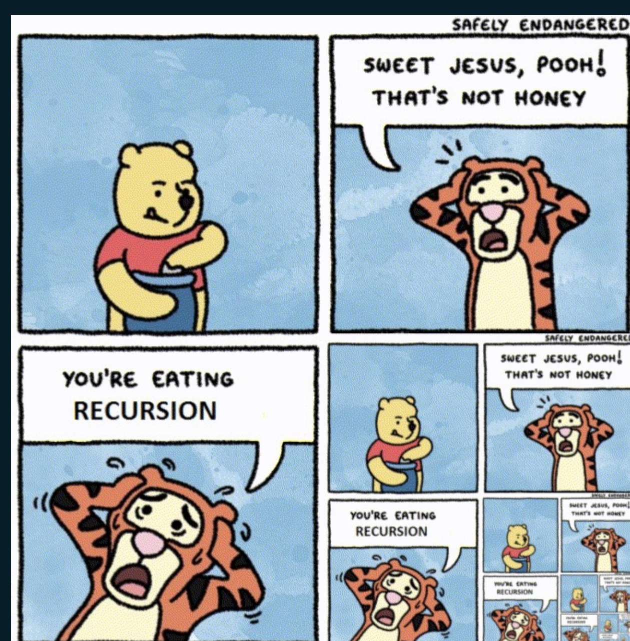
Function annotations

- To annotate a parameter:
  - Insert a colon after the parameter name, followed by the type
- To annotate the function with a return type:
  - Insert an arrow (dash and greater-than) and the type between the closing parameter(s) parenthesis and the function header colon :

```
def combat_round(char_a, char_b):
def combat_round(char_a: list, char_b: list) -> bool:
def foo(a: int, b: int = 5):
```


65

RECURSION



66

What is recursion?

A recursive function is a function that calls itself:

```
def main():
    message()

def message():
    print("This is my life now.")
    message( )
```


But that's just an infinite loop, Chris

- That's right!
- *A recursive function needs some way to control the number of times it gets called, just like a loop!*

```
def main():
    message(5)

def message(times):
    if times > 0:
        print("This is my life now.")
        message(times - 1)
```

What can recursion do?

- A problem can be solved with recursion if (and only if) it can be broken down into smaller problems that are identical in structure to the overall problem
- If the problem can be solved "now", without recursion, then the function solves the problem and returns the solution
- We call the problem we solve "now" the base case
- If the problem cannot be solved "now", then the function reduces the problem to a smaller but similar problem and calls itself to solve the smaller problem

Canonical example: factorial

```
def factorial(value):
    if value == 0:
        return 1
    else:
        return value * factorial(value – 1)
```

Here is a template

```
def recursive_function( parameter(s) ):

  if test for base case is true:
    # Provide base case answer

  else:
    # Do something with
    recursive_function(smaller argument(s))
```


I ❤️ recursion

1. Easy to understand
   - Base case
   - Recursive step
2. Clean clear pattern
   - If-else statement that gives us two choices
     1. Return the base case
     2. Or do a little something and make a recursive call
3. I think recursion is elegant

Greatest common divisor (GCD)

- The greatest common divisor of two numbers is the largest number that cleanly divides both numbers:
  1. The greatest common divisor of 6 and 9 is 3.
  2. The greatest common divisor of 5 and 10 is 5
  3. The greatest common divisor of 11 and 29 is 1.
- Observe the relationship between division and the remainder operator here
- In each case, the GCD is the largest number that leaves no remainder when we use % in Python with both values

Euclid to the rescue!

- If we can divide a number x by y and the remainder is 0, then y is the GCD of x and y
- This is our Base Case
- Otherwise, the GCD of x and y is the GCD of x and the remainder we just calculated
- Suppose we want to find the GCD of 270 and 192:

1. gcd(270, 192) 270 % 192 = 78 which is not 0, so this becomes
2. gcd(192, 78) 192 % 78 = 36 which is not 0, so this becomes
3. gcd(78, 36) 78 % 36 = 6 which is not 0, so this becomes
4. gcd(36, 6) 36 % 6 = 0 HOORAY so this returns 6.

Our implementation (try it!)

```
def gcd(x, y):
    if x % y == 0:
        return y
    else:
        return gcd(x, x % y)
```


Here is a more complex template

```
def recursive_function( parameter(s) ):
    optional statement block A
    if test for base case:
        provide base case answer
    else:
        optional statement block B
        call recursive_function(smaller argument(s))
        optional statement block C
```

Another example

```
def print_numbers(bound):
    if bound == 0:
        print(0)
    else:
        print(bound)
        print_numbers(bound – 1)
```


Another example

```
def print_triangle(base):
    if base == 0:
        return
    else:
        for x in range(base):
            print("*", end="")
        print()
        print_triangle(base - 1)
```

Another example

```
def spread_like_bacteria(location):
    if location is poisonous:
        die()
    else:
        colonize()
        spread_like_bacteria(up)
        spread_like_bacteria(down)
        spread_like_bacteria(left)
        spread_like_bacteria(right)
```


SOME CHALLENGES

1. Define a function called fibonacci(n) that uses recursion to find the nth value in the Fibonacci sequence
2. Define a function called is_palindrome(phrase) that uses recursion to determine if a phrase is a palindrome (return True) or not (return False)
3. Define a function called simple_exponent(base, exponent) that uses recursion to calculate and return base to the power of exponent.
   a) First assume base can be any value and exponent must be a positive integer
   b) Then modify the function so it permits negative integers as exponents.

ANOTHER CHALLENGE

1. Ackermann's Function is a famous recursive mathematical algorithm that can be used to test how well a system optimizes its performance using recursion.
2. Design a function called ackermann(m, n) which solves Ackermann's function. Use this logic in your function:
   1. If m == 0, then return n + 1
   2. If n == 0 then return ackermann(m - 1, 1)
   3. Otherwise, return ackermann(m - 1, ackermann(m, n - 1))

81

That's it for week 10!

This was a lot. I know. Thank you for working so hard. Please spend time working on A4 during the break between naps and games!

82