EPFL, CIVIL-127

Programming and software development for engineers

Indicative Student Feedback on Teaching

→ https://isa.epfl.ch

Revising previous labs

part 2

2.3: send + more = money

```
for s in range(1, 10):
   for e in range(10):
       if e == s:
           continue
       for n in range(10):
           if n == s or s == e:
               continue
           for d in range(10):
               if d == s or d == e or d == n:
                   continue
               ...
               send = 1000*s+100*e+10*n+d
               more = 1000*m+100*o+10*r+e
               money = 10000*m+1000*o+100*n+10*e+y
               if send + more == money:
                 print(send, more, money)
```

- We can find the solution using brute force (checking every possibility) – there are less than 2^27 possibilities
- Watch the start condition for the loops
  - `range(1, 10)` for `s` and `m`
  - `range(10)` for `e`, `n`, `d`, `o`, `r`, `y`
- Check if e.g. d has a given value with
  - `or` operator: `if d == s or d == e or d == n:`
  - Using a *set*: `if d in {s, e, n}:`
- Is this code clean, elegant, or *clever*?
  - No, but it works!
  - You can bruteforce upto 2^50-ish in a day

2.4: N people in a ring

```
def remove_every_k_element(N, K):
    active = K  # which person will be removed next
    people = list(range(N))

    while len(people) > 1:
        print("removing", people[active])
        people = people[:active]+people[active+1:]
        active = (active + K - 1) % len(people)
        print("remaining:", people)

remove_every_k_element(5, 2)
```

- Lists can be sliced with `list[start:end]`
- `+` on lists creates a new list with a copy of all the elements from both lists
- `people = people[:active] + people[active+1:]` to remove an element from the list
- Our solution is not very efficient. O(N) operation to remove each person, making it O(N^2) overall. We also need O(N) memory. We can do much better!

3.1: Stack

```
class Stack:
    def __init__(self):
        self.s = []

    def push(self, n):
        self.s.append(n)

    def pop(self):
        return self.s.pop()

    def max(self):
        if self.s == []:
            raise IndexError("empty stack")
        return max(self.s)
```

- Lists implement stack-like operations (pushing and popping)
- `max(list)` and `min(list)` are O(N) because they have to traverse the list

3.1: Stack in O1

```python
class StackO1:
    def __init__(self):
        self.s = []
        self.max_stack = []
        self.min_stack = []

    def push(self, n):
        self.s.append(n)
        new_max = n
        if self.max_stack != []:
            new_max = max(self.max(), n)
        self.max_stack.append(new_max)
    [...]
    def max(self):
        if self.s == []:
            raise IndexError("empty stack")
        return self.max_stack[-1]
```

- We could store `max`/`min` as attributes. `push()`/`max()`/`min()` would be O(1) operations but `pop()` would require recomputing the attributes (O(N)) if the value being popped equals the current max or min
- We could store `max`/`min` and `max_count`/`min_count` (to count how many times the max/min values have been seen, but we still have pop taking O(N) when the counts hit zero
- We can store the current `max`/`min` in their own stacks, all operations become O(1)

3.5: wrap_underscores

```
import unittest


def wrap_underscores(word):
    '''
    Adds an underscore between each letter.
E.g. "hello" becomes "_h_e_l_l_o_".
    If word is empty, returns an empty
string.
    '''
    if word == "":
        return ""
    r = "_"
    for i in range(len(word)):
        r += word[i] + "_"
    return r
```

- Forgetting a character (e.g. typing `=` instead of `+=`) is a common mistake, resulting in incorrect code
- Use your debugger, go step by step, and check if the variables have their expected value

3.5: wrap_underscores

```
import unittest


class
TestWrapUnderscores (unittest.TestCase):
   def test(self):
       a = wrap_underscores("hello")
       self.assertEqual(a, "_h_e_l_l_o_")


   def testEmpty(self):


self.assertEqual(wrap_underscores(""), "")
```

- When writing tests, think about edge cases, for strings
  - Empty string
  - Strings with repeated substrings
- For numbers
  - 0 value
  - Negative value
  - Large/boundary values

3.6: coins

```
def permutations(coinage, sum):
    solutions = []
    permutations2(coinage, sum, [],
solutions)
    for i, solution in enumerate(solutions):
        print("{}. {}".format(i+1, solution))

permutations([2, 3, 7], 40)
```

- `permutations` calls `permutations2` which does the actual search

3.6: coins

```
def permutations2(coinage, remaining, used,
solutions):
    if remaining == 0:
        # we have found a solution
        solutions.append(", ".join(used))
        return
    elif coinage == []:
        # we don't have any more coins left
        return
    ...
```

- This problem can be solved recursively
- If we have reached the desired sum, we have a solution
- If we don't have any more coins left, we are done
- Otherwise, we recurse with 0, 1, 2, … coins
- The maximum coins we can consider is `remaining // coin`

3.6: coins

```
    ...
    current_coin = coinage[0]
    max = remaining // current_coin

    for i in range(1, max+1):
        permutations2(coinage[1:], remaining
- i * current_coin, used + ["${} x
{}".format(current_coin, i)], solutions)

    # i=0 case
    permutations2(coinage[1:], remaining,
used, solutions)
```

- Otherwise, we recurse by taking 0, 1, 2, … coins
- The maximum coins we can consider is `remaining // current_coin`
- Why don't we print the solutions as we go?
  - We don't know if the solution we are considering is going to work or not!
- Be careful about mutating state when writing recursive functions
- This solution is inefficient, we might revisit it later

More Python Features

Statements and expressions

```
x = 1 + y + bar()


def foo():
    pass
```

- Statements
  - Assignments, loops, conditionals, function/class definitions
  - Have side effects (typically)
  - `x = 1 + y + bar()` is a statement
  - `pass` is a statement (placeholder)
- Expressions
  - Produces a value when evaluated
  - `1` is an expression (literal)
  - `y` is an expression (identifier)
  - `bar()` is an expression (function call)
  - `1 + y + bar()` is an expression

Statements and expressions

```
a = b = 1


while (value := input()) != "exit":
   print(f"You entered: {value}")
```

- This can become confusing!
  - Expressions are statements but not the other way
  - Chained assignments are statements
- `:=` is called the walrus operator and provides a way to have assignments inside expressions
- A piece of code is an expression if you can assign it or wrap it in `print(...)`
- Why isn't everything an expression?

Built-in data types

- Boolean
- Numbers (int, float, complex, and more)
- Strings
- Lists
- Tuple
- Dictionaries
- Sets

Boolean

- 2 values
  - `x = True`
  - `y = False`
- 3 operators
  - `x or y`
  - `x and y`
  - `not x`
- `or` evaluates second operand only if first is Falsy
- `and` evaluates second operand only if first is Truthy

Boolean: Falsy

- None
- 0, 0.0, 0j
- ""
- (), {}, [], set()
- range(0), ...
- Usually, the *empty* instance is considered Falsy (can be customized with `__bool__`)
- Considered `False` in an `if` or `while` statement

```
if []:
    print(1)
else:
    print(2) # prints 2
```

- Comparing values between different types usually results in False
  - `0 == []` is False
  - but `0 == False` is True

Boolean: Truthy

- 1, 2, 3, 1.0, 2.3, 1j
- "Foobar"
- (1)
- {1, 2, 3}
- [1, 2, 3]
- range(1)
- Considered `True` in an `if` or `while` statement

```
if 1.5:
    print(1) # prints 1
else:
    print(2)
```

- Comparing values between different types usually results in False
  - `1 == [1]` is False
  - but `1 == True` is True

Boolean

```
a = True or print(1)
b = True and print(2)
c = False or print(3)
d = False and print(4)
print("a:", a, "b:", b, "c:", c, "d:", d)
```

What's the output?

Boolean

```
a = True or print(1)
b = True and print(2)
c = False or print(3)
d = False and print(4)
print("a:", a, "b:", b, "c:", c, "d:", d)
```

Output:

```
2
3
a: True b: None c: None d: False
```

Boolean

```
def foo1():
    if bar():
        ...
        return True
    else:
        ...
        return False

def foo2():
    if bar():
        ...
        return True
    ...
    return False
```

- `foo1()` and `foo2()` are equivalent
- I prefer `foo2()` – less code, easier to read

Boolean: pro-tip

```
def foo(x):
    return x.imag or x


print(foo(4 + 2j))   # 2.0
print(foo(12))   # 12
print(foo(4 + 0j))   # (4+0j)
```

- Pro-tip: don't use `and` and `or` with non-boolean values – it'll come back and bite you
- Use conditional expressions instead, or just write longer but correct code
- In general, don't write "clever" code, follow the KISS principle (Keep It Simple Stupid) and write correct code
- This is a real world example which caused a bug

Numbers

- Integers
  - `x = 123 # decimal`
  - `x = 12_000_000 # decimal`
  - `x = 0x7b # hex`
  - `x = 0o173 # octal`
  - `x = 0b1111011 # binary`
- Floating point
  - `x = 45.6`
- Complex
  - `x = 7.8 + 0.9j`
- `fractions.Fraction`
- `decimal.Decimal`
- Lots of operators
  - `+`, `-`, `*`, `/`, `//`, `%`, `**`
  - And more with `import math`
- Bitwise operators
  - `|`, `^`, `&`, `<<`, `>>`, `~`

- `abs`, `bin`, `divmod`, `hex`, `oct`, `round`
  see https://docs.python.org/3.12/library/functions.html

Numbers

```
a = 0.1
b = 0.2
c = 0.3
print("1.", a + b == c)
print("2.", abs(a + b - c) < 0.001)
```

- Be careful about precision loss when doing floating point calculations!
- Outputs:
  1. `False`
  2. `True`

Operator precedence

```
print(3 < 4 == 2 < 3) # False
print(3<4 == 2<3) # False
print((3 < 4) == (2 < 3)) # True
print(3 < (4 == 2) < 3) # False
```

- See https://docs.python.org/3/reference/expressions.html#operator-precedence
- Operator precedence rules differ between programming languages, careful if you are porting code
- When in doubt, use parenthesis

Dictionaries

- Key-value data structure
- `d = {"foo": 123, "bar": 567}`
- Empty dictionary
  - `d = {}`
- Dictionary comprehension
  - `d = {x[0]: x for x in ["foo", "bar"]}`
  - Equivalent to `d = {'f': 'foo', 'b': 'bar'}` in this example

Sets

- Mutable data structure
- `s = set("hello")` or `s = {"e", "h", "l", "o"}`
- `s.add()`, `s.remove()`, and `s.discard()`
- `"x" in s`, `"x" not in s`
- `isdisjoint()`
- `issubset()`, `<=`, `<`
- `issuperset()`, `>=`, `>`
- `union()`, `|`
- `intersection()`, `&`
- `difference()`, `-`
- `symmetric_difference()`, `^`
- Empty set
  - `set()`

Sets

```
s = "hello"

if len(s) == len(set(s)):

    ...
```

- `set()` takes an iterable
- Use `len()` on a string and its conversion to a set
  - Checks if there are any repeated characters

Functions/methods

```
def foo(a, b):
    print("a:", a, "b:", b)


foo(1, 2)
foo(a=1, b=2)
foo(b=2, a=1)
foo(1, b=2)
```

- You can call functions by respecting the parameter order (positional argument)
- You can call functions by providing the parameter name (keyword argument)
- You can mix the two, but positional arguments must come first
- Implicit `return None` at the end

Functions/methods: default values

```
def foo(a, b, c=99):
    print("a:", a, "b:", b, "c:", c)

foo(1, 2)
foo(1, 2, 3)
foo(a=1, b=2)
foo(b=2, a=1, c=3)
foo(1, b=2, c=3)
```

- Function parameters can have default values
- E.g. with `print()`, there's are default behaviors you can override – `sep=' '`, `end="\n"`…

Functions/methods: default values

```
def foo(a, b=[]):
    b.append(a)
    return b

x = foo(1, [])
y = foo(2)
z = foo(3)
print(x, y, z)   # [1] [2, 3] [2, 3]
```

- Be very careful if a default value is mutable
- Mutations might have unintended consequences since there's only one copy

Functions/methods: variadic

```
def foo(*args):
    s = 0
    for i in args:
        s += i
    return s


print(foo(1, 2, 3, 4, 5))  # 15
```

- Functions can take a variable number of positional parameters, that's how `print()` works
- `args` becomes a tuple, you can iterate on it or call `len`

Functions/methods: variadic

```
def foo(**args):
    for k, v in args.items():
        print(k, v)



foo(a=1, b=2, c=3, d=4, e=5)
```

- Functions can take a variable number of keyword parameters, that's how `print()` works
- `args` becomes a dict

/ and * in function/methods definitions

```
def combined_example(pos_only, /, standard, *,
kwd_only):
   print(pos_only, standard, kwd_only)
```

- If / and * are not present in the function definition, arguments may be passed to a function by position or by keyword
- A slash in the argument list of a function denotes that the parameters prior to it are positional-only
- A star in the argument list of a function denotes that the parameters after it are keyword-only
- Don't confuse the star by itself with variadic

Functions/methods: decorators

```
import datetime

def time_of_day_greeting(func):
    def wrapper(name):
        if datetime.datetime.now().hour < 12:
            func("Good morning " + name)
        else:
            func("Hello " + name)
    return wrapper

@time_of_day_greeting
def greet(name):
    print(name)

greet("Joe")  # Good morning Joe
# or Hello joe, depending on the computer's clock
```

- `@decorator_name` above function/method definition
- Decorators enable intercepting the execution of functions
- Creating decorators is difficult (typically needs to work with any function), so you'll rarely create decorators but you might use them
- A function/method can have multiple decorators

Functions/methods: decorators

```
from functools import cache


@cache
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)


fib(36)
```

- @cache is a useful decorator
  - It will record the function's arguments and response. If a future call is made with the same arguments, the response is returned from memory storage instead of being re-computed
  - It's yet another CPU⇔memory tradeoff
- Without caching, this code takes ~1.5 seconds to run
- With caching, it takes <1ms

Function assignment

```
def foo():
    print("foo")


def bar():
    print("bar")


x = input("call foo? ")
if x == "y":
    y = foo
else:
    y = bar
y()
```

- You can assign functions to variables and call them later…
- …but you **must** favor OOP over this kind of dynamic dispatch

Function inside function

```
def foo(a, b):
    def bar(c):
        print(a, b, c)
    bar(3)

foo(1, 2)
```

- You can define a function inside another function
- But don't
  - Testing is harder
  - Can't reuse the function from outside
  - Use classes to group related functions

Lambdas

```
l = sorted([(1, "blue"),
            (4, "red"),
            (10, "green")],
           key=lambda x: len(x[1]))
print(l)  # [(4, 'red'), (1, 'blue'),
(10, 'green')]
```

- Lambdas are anonymous functions with an implied return
- Useful e.g. with `sort` and `sorted` to customize the sorting behavior

Lambdas

```
# anti-patterns
normalize_case = lambda s: s.casefold()

l = sorted(["blue", "red", "green"],
  key=len)
print(l) # ['red', 'blue', 'green']
```

- Don't use
  - If you are going to assign the lambda to a variable. Use a regular function definition instead
  - Use a lambda when you can simply use the function name (see operators module)
  - If writing the code on multiple lines will be easier to read
- Lambdas generally hurt
  - Readability
  - Testability
  - Reusability
- So weigh the pros/cons before using them

Lambdas

```
s = []
for x in range(5):
    s.append(lambda: x**2)

for x in range(5):
    s.append(lambda n=x: n**2)

for i in s:
    print(i())
```

- Lambdas capture scope, you can control when the variable will get bound
- Output:

```
16
16
16
16
16
0
1
4
9
16
```

Control Flow

- if
- for
- while
- match

Control flow enables automation – the core reason for writing programs.

if, elif, elif, elif, else

```
if some_condition:
    # go left
    ...
elif some_other_condition:
    # go right
    ...
elif yet_another_condition:
    # do something
    ...
else:
    # reject
    ...
```

- Comment your code inside the if/elif/else to remove redundancy
- You can have multiple `elif` statements

if, elif, elif, elif, else

```
def foo():
    if some_condition:
        ...
        return

    # don't need an else case here
    if some_other_condition:
        ...
        return

    ...
```

- When you `return` from the first `if` block, you don't need an `elif` or `else` block.

conditional expression

```
a = 1
b = "foo" if a == 1 else "bar"
print(b) # foo
```

- Some people like this feature

Nested loops

```
for i in range(10):
    for j in range(10):
        if j == i:
            continue
        if j + i == 10:
            break
        print(i, j)
    if i % 3 == 0:
        continue
```

- `break` and `continue`, to break or continue iterating
- You can't break or continue the outer loop from within the inner loop, you must refactor your code

Nested loops

```
def foobar():
    for i in range(10):
        for j in range(10):
            if j == i:
                continue
            if j + i == 10:
                return
            print(i, j)
        if i % 3 == 0:
            continue
```

- You can't break or continue the outer loop from within the inner loop, you must refactor your code and use a `return` statement

while loops

```
def prompt_yes_or_no(prompt=""):
    while True:
        a = input(prompt)
        if a == "yes" or a == "no":
            return a
```

- If you know how many times you want to loop, use `for`
- Otherwise, use `while`
- You can use `break` and `continue`, like you would with `for` loops

match

```
match input():
    case "w":
        ...
    case "d":
        ...
    case "a":
        ...
    case "s":
        ...
```

- Replaces many `if`, `elif`, `elif`, `else` statements

match

```
match input():
    case "w" | "i":
        ...
    case "d" | "k":
        ...
    case "a" | "j":
        ...
    case "s" | "l":
        ...
```

- Combine multiple cases, replacing `or` expressions

match

```
a = ((1, 2), "foo")

match a:
    case ((1, _), "bar"):
        print("1.")
    case ((x, 2), "foo"):
        print("2.", x)
    case _:
        print(a)
```

- Destructure tuples, first case wins
- Use `case _:` to handle the default case
- More info https://peps.python.org/pep-0636/