Design

Class outline:

- Functional abstractions
- What's in a name?
- f-strings
- Debugging & errors

Functional abstractions

Abstraction

In CS, we often "abstract away the details":
We intentionally ignore some details in order to provide a consistent interface.

Abstraction by parameterization

In a world before functions...

```
interest =  1 + 0.6 * 2
interest2 =  1 + 0.9 * 4
interest3 =  1 + 2.1 * 3
```

Parameterized!

```
def interest(rate, years):
    return 1 + rate * years
```

A **parameterized function** performs a computation that works for all acceptable values of the parameters.

✂ Removed detail: the values themselves!

Abstraction by specification

A specification for the built-in `round` function:

> `round(number[, ndigits])`: Return number rounded to `n` digits precision after the decimal point. If `n` digits is omitted or is `None`, it returns the nearest integer to its input.
> See full documentation.

A well-designed **function specification** (function signature + docstring) serves as a contract between the implementer and the user.

✂ Removed detail: the implementation!

Using an abstraction

Based on this specification..

`square(n)`: Returns the square of the number `n`.

This should work!

```
def sum_squares(x, y):
    """
    >>> sum_squares(3, 9)
    90
    """
    return square(x) + square(y)
```

Implementing the abstraction

Many possible implementations can be used:

Implementing the abstraction

Many possible implementations can be used:

```
def square(x):
    return pow(x, 2)
```

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

```
from operator import mul

def square(x):
    return mul(x, x)
```

```
square = lambda x: x * x
```

It could even be built-in to Python, in theory!

Not all implementations are equal

An implementation may have practical consequences:

- Affecting the size of the program
- Affecting the speed of the program's execution

Not the ideal implementation:

```
from operator import mul

def square(x):
    return mul(x, x-1) + x
```

But you can cross that bridge when you come to it.

What's in a name?

There are only two hard things in Computer Science: cache invalidation and naming things. --Phil Karlton

Choosing names

Names typically don't matter for correctness but they matter a lot for readability.

From 😟	To
`true_false`	`rolled_one`
`d`	`dice`
`helper`	`take_turn`
`my_int`	`num_rolls`

Names should convey the meaning or purpose of the values to which they are bound.

Function names typically convey their effect (`print`), their behavior (`triple`), or the value returned (`abs`).

Parameter names

The type of value bound to a parameter name is best documented in a function's docstring.

```
def summation(n, f):
    """Sums the result of applying the function F
    to each term in the sequence from 1 to N.
    N can be any integer > 1, F must take a single
    integer argument and return a number.
    """
    total = 0
    k = 1
    while k <= n:
        total = total + f(k)
        k = k + 1
    return total
```

Which values deserve a name?

Repeated compound expressions:

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

Which values deserve a name?

Repeated compound expressions:

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

Meaningful parts of complex expressions:

```
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)
```

More naming tips

Names can be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc.

- `n`, `k`, `i` - Usually integers
- `x`, `y`, `z` - Usually real numbers or coordinates
- `f`, `g`, `h` - Usually functions

More naming tips

Names can be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc.

- `n`, `k`, `i` - Usually integers
- `x`, `y`, `z` - Usually real numbers or coordinates
- `f`, `g`, `h` - Usually functions

Names can be long if they help document your code:

```
average_age = average(age, students)
```

is preferable to...

```
# Compute average age of students
aa = avg(a, st)
```

String formatting

String concatenation

So far, we've been using the + operator for combining string literals with the results of expressions.

```
artist = "Lil Nas X"
song = "Industry Baby"
place = 2

print("Debuting at #" + str(place) + ": '" + song + "' by " + artist)
```

But that's not ideal:

- Easy to bungle up the + signs
- Hard to grok what the final string will be
- Requires explicitly `str()`ing non-strings

String interpolation

String interpolation is the process of combining string literals with the results of expressions.

Available since Python 3.5, **f strings** (formatted string literals) are the best way to do string interpolation.

Just put an `f` in front of the quotes and then put any valid Python expression in curly brackets inside:

```
artist = "Lil Nas X"
song = "Industry Baby"
place = 2

print(f"Debuting at #{place}: '{song}' by {artist}")
```

😍😍😍😍😍😍

Expressions in f strings

Any valid Python expression can go inside the parentheses, and will be executed in the current environment.

```
greeting = 'Ahoy'
noun = 'Boat'

print(f"{greeting.lower()}, {noun.upper()}yMc{noun}Face")

print(f"{greeting*3}, {noun[0:3]}yMc{noun[-1]}Face")
```

Errors

Types of errors

These are common to all programming languages:

- Logic errors
- Syntax errors
- Runtime errors

Logic errors

Logic errors

A program has a logic error if it does not behave as expected. Typically discovered via failing tests or bug reports from users.

Spot the logic error:

```
# Sum up the numbers from 1 to 10
sum = 0
x = 1
while x < 10:
    sum += x
    x += 1
```

To avoid the wrath of angry users, write tests.

Syntax errors

Syntax errors

Each programming language has syntactic rules. If the rules aren't followed, the program cannot be parsed and will not be executed at all.

Spot the syntax errors:

```
if x > 5
   x += 1
```

```
sum = 0
x = 0
while x < 10:
   sum + = x
   x + = 1
```

To fix a syntax error, read the message carefully and go through your code with a critical eye.

Syntax errors

Each programming language has syntactic rules. If the rules aren't followed, the program cannot be parsed and will not be executed at all.

Spot the syntax errors:

```
if x > 5 # Missing colon
   x += 1
```

```
sum = 0
x = 0
while x < 10:
   sum + = x
   x + = 1
```

To fix a syntax error, read the message carefully and go through your code with a critical eye.

Syntax errors

Each programming language has syntactic rules. If the rules aren't followed, the program cannot be parsed and will not be executed at all.

Spot the syntax errors:

```
if x > 5 # Missing colon
   x += 1
```

```
sum = 0
x = 0
while x < 10:
   sum + = x # No space needed between + and =
   x + = 1
```

To fix a syntax error, read the message carefully and go through your code with a critical eye.

SyntaxError

What it technically means:
The file you ran isn't valid python syntax

What it practically means:
You made a typo

What you should look for:

- Extra or missing parenthesis
- Missing colon at the end of an if, while, def statements, etc.
- You started writing a statement but forgot to put any clauses inside

Examples:

```
print("just testing here"))
```

```
title = 'Hello, ' + name ', how are you?'
```

`IndentationError`/`TabError`

What it technically means:
The file you ran isn't valid Python syntax, due to indentation inconsistency.

What it sometimes means:
You used the wrong text editor (or one with different settings)

What you should look for:

- A typo or misaligned block of statements
- A mix of tabs and spaces
  - Open your file in an editor that shows them
  - `cat -A filename.py` will show them

Example:

```
def sum(a, b):
    total = a + b
        return total
```

Runtime errors

Runtime errors

A runtime error happens while a program is running, often halting the execution of the program. Each programming language defines its own runtime errors.

Spot the runtime error:

```
def div_numbers(dividend, divisor):
    return dividend/divisor

quot1 = div_numbers(10, 2)
quot2 = div_numbers(10, 1)
quot3 = div_numbers(10, 0)
quot4 = div_numbers(10, -1)
```

To prevent runtime errors, code defensively and write tests for all edge cases.

Runtime errors

A runtime error happens while a program is running, often halting the execution of the program. Each programming language defines its own runtime errors.

Spot the runtime error:

```
def div_numbers(dividend, divisor):
    return dividend/divisor

quot1 = div_numbers(10, 2)
quot2 = div_numbers(10, 1)
quot3 = div_numbers(10, 0)  # Cannot divide by 0!
quot4 = div_numbers(10, -1)
```

To prevent runtime errors, code defensively and write tests for all edge cases.

`TypeError:'X' object is not callable`

What it technically means:
Objects of type X cannot be treated as functions

What it practically means:
You accidentally called a non-function as if it were a function

What you should look for:

- Parentheses after variables that aren't functions

Example:

```
sum = 2 + 2
sum(3, 5)
```

...NoneType...

What it technically means:
You used None in some operation it wasn't meant for

What it practically means:
You forgot a return statement in a function

What you should look for:

- Functions missing return statements
- Printing instead of returning a value

Example:

```
def sum(a, b):
    print(a + b)

total = sum( sum(30, 45), sum(10, 15) )
```

NameError

What it technically means:
Python looked up a name but couldn't find it

What it practically means:

- You made a typo
- You are trying to access variables from the wrong frame

What you should look for:

- A typo in the name
- The variable being defined in a different frame than expected

Example:

```
fav_nut = 'pistachio'
best_chip = 'chocolate'
trail_mix = Fav_Nut + best__chip
```

UnboundLocalError

What it technically means:
A variable that's local to a frame was used before it was assigned

What it practically means:
You are trying to both use a variable from a parent frame, and have the same variable be a local variable in the current frame

What you should look for:
Assignments statements after the variable name

Example:

```
sum = 0

def sum_nums(x, y):
    sum += x + y
    return sum

sum_nums(4, 5)
```

TraceBacks

What's a traceback?

When there's a runtime error in your code, you'll see a **traceback** in the console.

```
def div_numbers(dividend, divisor):
    return dividend/divisor

quot1 = div_numbers(10, 2)
quot2 = div_numbers(10, 1)
quot3 = div_numbers(10, 0)
quot4 = div_numbers(10, -1)
```

```
Traceback (most recent call last):
    File "main.py", line 14, in <module>
        quot3 = div_numbers(10, 0)
    File "main.py", line 10, in div_numbers
        return dividend/divisor
ZeroDivisionError: division by zero
```

Parts of a Traceback

- The error message itself
- Lines #s on the way to the error
- What's on those lines

The most recent line of code is always last (right before the error message).

```
Traceback (most recent call last):
    File "main.py", line 14, in <module>
        quot3 = div_numbers(10, 0)
    File "main.py", line 10, in div_numbers
        return dividend/divisor
ZeroDivisionError: division by zero
```

Reading a Traceback

1. Read the error message (remember what common error messages mean!)
2. Look at each line, bottom to top, and see if you can find the error.

```
Traceback (most recent call last):
    File "main.py", line 14, in <module>
        quot3 = div_numbers(10, 0)
    File "main.py", line 10, in div_numbers
        return dividend/divisor
ZeroDivisionError: division by zero
```

Fix this code!

```
def f(x):
    return g(x - 1)

def g(y):
    return abs(h(y) - h(1 /& y)

def h(z):
    z * z

print(f(12))
```