Using Objects

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Poll: What gets printed?

```
class Cat:
    def __init__(self, name: str, human: str):
        self.name = name
        self.human = human
        print(name * 3)

mini: Cat = Cat('Mini', 'Rasika')
```

1. `Mini`
2. `<__main__.Cat object at 0x10d380200>`
3. `MiniMiniMini`
4. (Nothing)

State and aliasing

When a variable holds an object, it holds a *reference* to that object. That object is stored somewhere in the memory of the computer, and the variable knows where to access it.

Multiple variables can hold references to the same object.

Alias: a second reference to an existing object (same place in computer's memory)

Copy: another object that looks exactly the same (different place in computer's memory)

```
original: list[int] = [1, 2, 3]

alias: list[int] = original
copy: list[int] = original.copy()

original[2] = 90

print(alias) # [1, 2, 90]
print(copy) # [1, 2, 3]
```

Passing mutable objects as arguments

If we pass an object as an argument to a function, an *alias* is created, not a copy.

```
def sum_with_bad_manners(in_list: list[int]) -> int:
    sum: int = 0
    while (len(in_list) > 0):
        sum += in_list.pop()
    return sum


my_list: list[int] = [1, 2, 3, 4]
print(f'Sum: {sum_with_bad_manners(my_list)}') # Sum: 10
print(my_list) # []
```

Good manners: Functions should leave their args unchanged (unless they clearly state otherwise in the documentation).

```python
class Shirt:
    def __init__(self, size: int):
        self.size = size

def main() -> None:
    six_hundred: int = 600
    subtract_one(six_hundred)
    print(six_hundred)                  # 600

    shirt: Shirt = Shirt(600)
    shrink_shirt(shirt)
    print(shirt.size)                   # 599

def subtract_one(num: int) -> None:
    num -= 1

def shrink_shirt(shirt: Shirt) -> None:
    shirt.size -= 1
```

Poll: Which of these makes it print 4?

```
def main() -> None:
    shirt: Shirt = Shirt(600)
    modify_shirt(shirt)
    print(shirt.size)

if __name__ == '__main__':
    main()
```

1.

```
def modify_shirt(shirt: Shirt) -> None:
    shirt.size = 4
```

2.

```
def modify_shirt(shirt: Shirt) -> None:
    shirt = Shirt(4)
```

3.

```
def modify_shirt(shirt: Shirt) -> Shirt:
    return Shirt(4)
```

The `__str__()` function

Every class has a `__str__()` method.

We can overwrite it with our own `__str__()` method.

When we print an object, it implicitly calls the `__str__()` method.

The default `__str__()` method is not helpful.

```
mini = Cat('Mini', 'Rasika')
print(mini) # <__main__.Cat object at 0x1095ca790
```

The `__str__()` function

Why is it different when we print a list (which is also an object)?
We usually write our own `__str__()` method that returns a more helpful `str` for that class.

```
class Cat:
    ...
    def __str__(self):
        return f'{self.name} meows to {self.human}'


mini = Cat('Mini', 'Rasika')
print(mini) # Mini meows to Rasika
```

Poll: What is printed?

```
class Cat:
    def __init__(self, name: str, human: str):
        self.name = name
        self.human = human

    def __str__(self) -> str:
        print('MUAHAHAHAHHA')
        return self.name

mini: Cat = Cat('Mini', 'Rasika')
print(mini)
```

1. `Mini`
2. `MUAHAHAHAHHA` // `Mini`
3. `<__main__.Cat object at 0x10d380200>`
4. `MUAHAHAHAHHA` // `<__main__.Cat object at 0x10d380200>`

Generic types

In Python:

We are able to put objects of different types into the same `list`, but it's discouraged (makes the list harder to process).

In CS2100:

Elements of a list must be of the same type since we require types in our Python code. What would be the type of the variable `my_list = [1, 'a']`?

The same `list` class can make objects of different types (`list[str]` and `list[int]`) because `list` is a *generic* type.

Define our own generic type:

1. First define the type variable `T`
2. Then use `T` to define the generic type `Stack[T]`
3. Inside the class `Stack[T]`, the `T` can be any type, but all instances of `T` must be the same type as each other
4. Instantiate it as `my_stack`, with `T` taking the value `int`
5. Instantiate another variable `my_other_stack`, where `T` is `str`

```python
from typing import TypeVar, Generic

T = TypeVar('T')

class Stack(Generic[T]):
    def __init__(self) -> None:
        self.items: list[T] = []

    def push(self, item: T) -> None:
        self.items.append(item)

    def pop(self) -> T:
        return self.items.pop()

    def is_empty(self) -> bool:
        return not self.items

my_stack: Stack[int] = Stack()
my_stack.push(4)
print(my_stack.pop())   # 4
my_other_stack: Stack[str] = Stack()
```

Definitions

- **Generic type**: a class with a type variable, like `list[T]`
- **Parameterized type**: a generic type with the type variables filled in, like `list[str]`
- **Raw type**: a generic type without the type variable, like `list`
    - use this if we don't need to re-use the type variable anywhere else in the code

We can parametrize the type using another user-defined type:

```
stack_of_stacks: Stack[Stack[int]] = Stack()
```

Poll: Which of these is allowed?

```
class Thing(Generic[T]):
    def __init__(self, item: Optional[T]):
        """Item is of type T or None"""
        self.item = item
```

1. `item: Thing[str] = Thing('hello')`
2. `item: Thing[str] = Thing(None)`
3. `item: Thing[str] = Thing(5)`
4. `item: Thing[Thing[str]] = Thing(Thing('hello'))`

Functions with lots of arguments:

```
def display_text(
    text: str, size: int, is_bold: bool,
    is_italic: bool, is_underlined: bool) -> None:
    ...

display_text('hello', 18, False, False, False)
display_text('goodbye', 18, True, False, False)
```

- **Pros**: multiple options in the same function without compromising flexibility
- **Cons**: error prone, must keep track of order of arguments, too many things to specify each time we call the function

Two solutions: named args and default arg values

Named arguments

```
def display_text(
    text: str, size: int, is_bold: bool,
    is_italic: bool, is_underlined: bool) -> None:
    ...

display_text(
    text = 'hello', is_underlined = False,
    is_bold = False, is_italic = False, size = 18)
```

- Make calls more readable
- Enable you to reorder arguments

Default argument values

```
def display_text(
    text: str, size: int = 18, is_bold: bool = False,
    is_italic: bool = False, is_underlined: bool = False
) -> None:
    ...

display_text(text = 'hello', is_bold = True)
```

- If you usually pass the same value
- Default value when the client doesn't specify a value when calling it
- Args with default values must come after args without default values
- If we have a function that is already widely used, and we want to add another parameter, give it a default value (so the existing code doesn't break)

Default arg values are evaluated when the function is declared, not when it is called.

It is stuck with the value it got the first time.

It does not "refresh" each time the function is called.

```
number = 5

def print_number(number: int = number) -> None:
    print(number)

number = 6 # This line does nothing
# Default value for the argument is stuck at 5

print_number(8) # 8
print_number()  # 5
print(number)   # 6
```

Poll: What does this output? Why?

```
def send_message_and_cc_self(
        message: str, sender: str, recipients: list[str] = []) -> None:

    recipients.append(sender) # add sender to recipients

    for r in recipients:
        print(f"Sending '{message}' from {sender} to {r}")

send_message_and_cc_self("note to self", "Rasika")
send_message_and_cc_self("use RSA next time", "Eve", ["Alice", "Bob"])
send_message_and_cc_self("super secret", "admin")
```

This is an example of code that could look like it's doing one thing, when it's actually doing something else

Source: Tyler Yeats

Variable argument lists

Python allows us to have a function with an arbitrary number of arguments:

```
def print_args(*args: T) -> None:
    """Print each argument on a separate line"""
    for item in args:
        print(item)

print_args(1, 2, 3)
```

- `print_args()` can take any number of arguments
- They are of type `T`
- We can access them inside the function
  - each arg is an element in the tuple `args`
  - if there are no args, then `args` will be an empty tuple

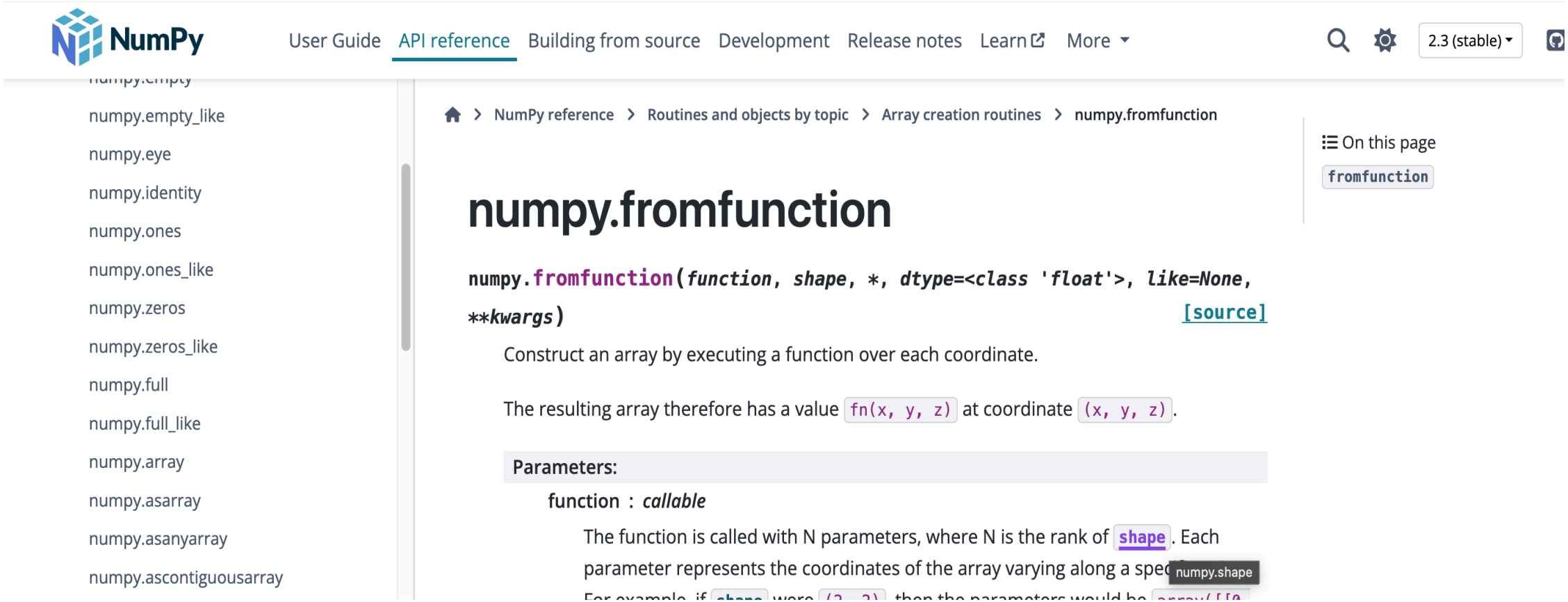
Variable argument list, but with named arguments:

```
def print_args(**kwargs: T) -> None:
    """Print each argument on a separate line"""
    for argument_name, argument_value in kwargs.items():
        print(f'{argument_name}: {argument_value}')

print_args(a = 1, b = 2, c = 3)

a: 1
b: 2
c: 3
```

`**kwargs` stands for "keyword arguments", but you can name it anything you want.

We use two asterisks for `**kwargs` and one for `*args`.

Poll: How many arguments can I pass to this function?



a. 0
b. 1
c. 2
d. 3
e. 4
f. 6
g. 10

The Single Responsbility Principle

A core principle for writing code is that every component of our code must have a single purpose. This makes our code easier to read, test, and maintain.

Component	Does not Follow Single Responsibility Principle
Variable	`age_or_nonexistent: int = -1 # negative if nonexistent`
Function	`def read_file_compute_average_print_score(filename: str) -> None:`
Class	`class FileManagerAndOutputFormatter`

Abstraction

We organize lines of code into functions, functions into classes, classes into...

"Absraction": the prodecure of grouping more granular things into less granular groups

Benefits of abstraction we already saw in functions:

- reuse code without redundancy
- hide implementation details
- break down a problem into smaller, more manageable pieces

The same applies to the idea of putting methods into classes.

Abstraction

Hard to debug or modify:

```
length: int = 5
width: int = 3

print(get_area_of_rectangle(
    length, width))

length = 6
width = 4

print(get_perimeter_of_rectangle(
    length, width))
```

Put relevant perimeter and area methods into a class for each shape:

```
table: Rectangle = Rectangle(5, 3)
print(table.area())

print(Rectangle(6, 4).perimeter())

chair: Square = Square(4)
print(chair.area())
```

Benefits of abstraction:

- We can use the same code multiple times without re-writing it
- It's easier to read
- It's easier to maintain and adapt the code later on
- It's easier to test behaviors in isolation
- Each variable, function, and class has a single, clear responsibility

Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?