Generics and Changing Arguments

Welcome back to CS 2100!

Prof. Rasika Bhalerao

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 List, 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'))`

Generic functions

A function can take a generic type as an argument:

```
def get_first(list: List[T]) -> T:
    if len(list) > 0:
        return list[0]
    else:
        raise ValueError
```

Note: We didn't have to redefine `T` -- we only need to define it once at the top, and we can reuse it.

Note: If we need two different type variables (to specify that the other type can be different from `T`), then we will need to define a second type variable.

Exercise: `map()`

Let's write a function `map()` that converts a list of one type into a list of another type.

Its arguments should be:

- a list of generic type `T`
- a function that takes `T` and returns `R` (another type variable)

Hint: the type for a function that takes `T` and returns `R` is `Callable[[T], R]`

Solution

```
def map(original: List[T], mapper: Callable[[T], R]) -> List[R]:
    """Returns a copy of the list containing elements converted using the mapper

    Parameters
    ----------
    original : List[T]
        The original list
    mapper: Callable[[T], R]
        The function to convert elements from the original list to the new list

    Returns
    -------
    List[R]
        A new list with the mapped elements
    """
    return [mapper(i) for i in original]
```

Poll: Which function can we NOT pass to `map(original: List[T], mapper: Callable[[T], R]) -> List[R]`?

1. `def to_str(inp: int) -> str:`
2. `def add_one(inp: int) -> int:`
3. `def to_int_or_None(inp: str) -> Optional[int]:  # returns int or None`
4. `def add(inp1: int, inp2: int) -> int`

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

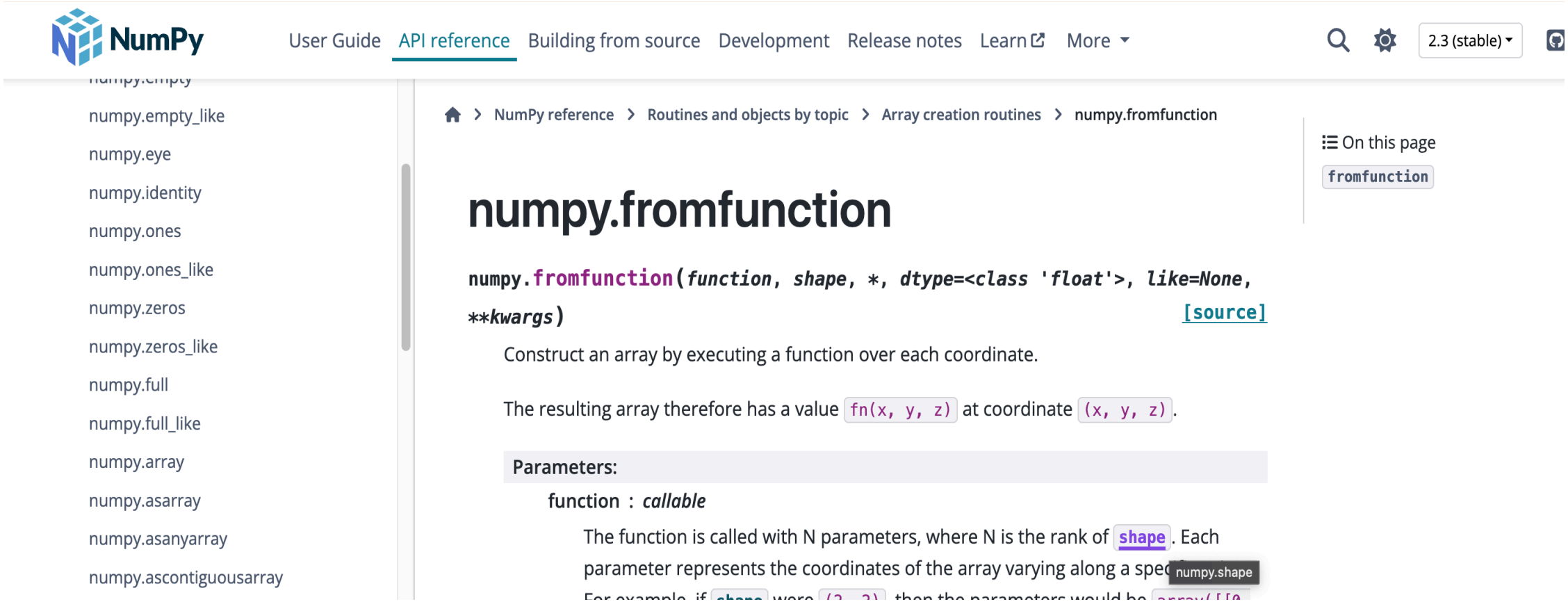
Variable argument list, but with named arguments:

```
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

Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?