

Representation

Class outline:

- String formatting
- repr/str representation
- Special method names
- Polymorphism
- Generics

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")
```

Objects

So many objects

What are the objects in this code?

```
class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

    def __init__(self, name):
        self.name = name

    def play(self):
        self.happy = True

lamb = Lamb("Lil")
owner = "Mary"
had_a_lamb = True
fleece = {"color": "white", "fluffiness": 100}
kids_at_school = ["Billy", "Tilly", "Jilly"]
day = 1
```


So many objects

What are the objects in this code?

```
class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

    def __init__(self, name):
        self.name = name

    def play(self):
        self.happy = True

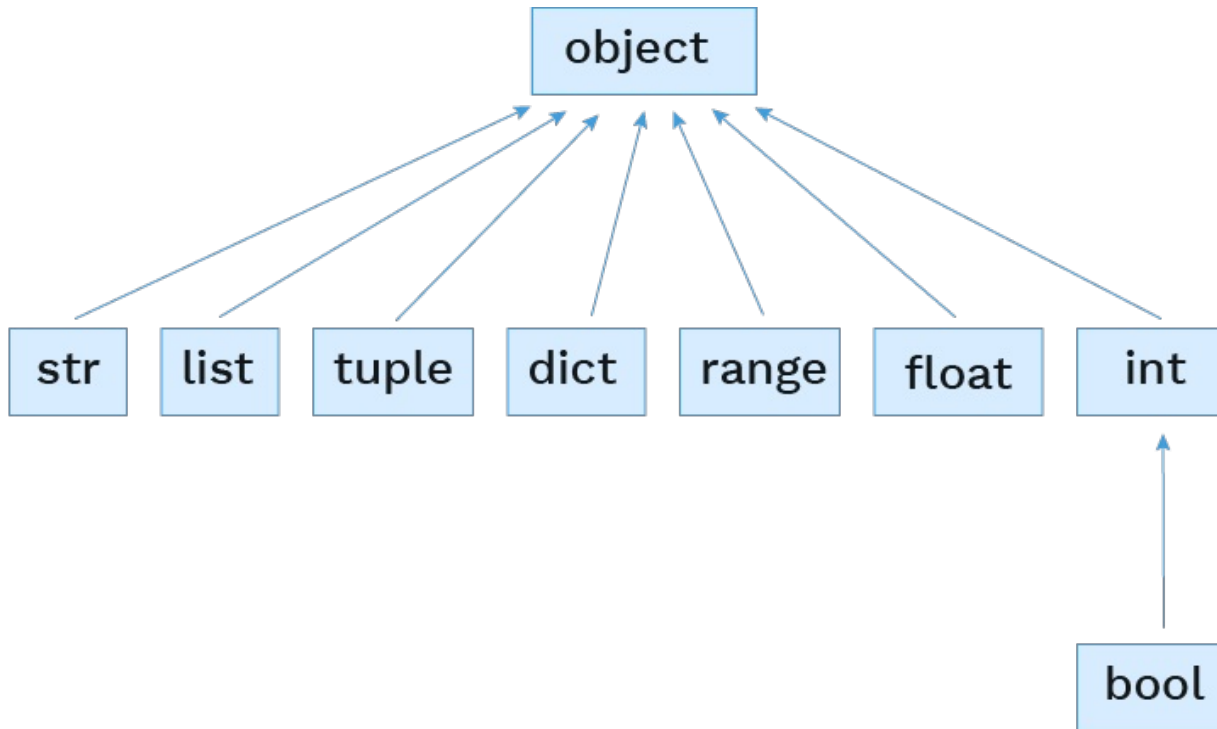
lamb = Lamb("Lil")
owner = "Mary"
had_a_lamb = True
fleece = {"color": "white", "fluffiness": 100}
kids_at_school = ["Billy", "Tilly", "Jilly"]
day = 1
```

`lamb`, `owner`, `had_a_lamb`, `fleece`, `kids_at_school`, `day`, etc.

We can prove it by checking `object.__class__.__bases__`, which reports the base class(es) of the object's class.

It's all objects

All the built-in types inherit from `object`:



Built-in object attributes

If all the built-in types and user classes inherit from `object`, what are they inheriting?

Just ask `dir()`, a built-in function that returns a list of all the attributes on an object.

```
dir(object)
```

Built-in object attributes

If all the built-in types and user classes inherit from `object`, what are they inheriting?

Just ask `dir()`, a built-in function that returns a list of all the attributes on an object.

```
dir(object)
```

- For string representation: `__repr__`, `__str__`, `__format__`
- For comparisons: `__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__ne__`
- Related to classes: `__bases__`, `__class__`, `__new__`, `__init__`, `__init_subclass__`, `__subclasshook__`, `__setattr__`, `__delattr__`, `__getattr__`
- Others: `__dir__`, `__hash__`, `__module__`, `__reduce__`, `__reduce_ex__`

Python calls these methods behind these scenes, so we are often not aware when the "dunder" methods are being called.

Let us become enlightened!

String representation

__str__

The `__str__` method returns a human readable string representation of an object.

```
from fractions import Fraction
```

```
one_third = 1/3
```

```
one_half = Fraction(1, 2)
```

```
float.__str__(one_third)
```

```
Fraction.__str__(one_half)
```

__str__

The `__str__` method returns a human readable string representation of an object.

```
from fractions import Fraction
```

```
one_third = 1/3
```

```
one_half = Fraction(1, 2)
```

```
float.__str__(one_third)      # '0.3333333333333333'
```

```
Fraction.__str__(one_half)   # '1/2'
```

__str__ usage

The `__str__` method is used in multiple places by Python: `print()` function, `str()` constructor, f-strings, and more.

```
from fractions import Fraction
```

```
one_third = 1/3
```

```
one_half = Fraction(1, 2)
```

```
print(one_third)
```

```
print(one_half)
```

```
str(one_third)
```

```
str(one_half)
```

```
f"{one_half} > {one_third}"
```


__str__ usage

The `__str__` method is used in multiple places by Python: `print()` function, `str()` constructor, f-strings, and more.

```
from fractions import Fraction
```

```
one_third = 1/3
```

```
one_half = Fraction(1, 2)
```

```
print(one_third)           # '0.3333333333333333'
```

```
print(one_half)           # '1/2'
```

```
str(one_third)            # '0.3333333333333333'
```

```
str(one_half)            # '1/2'
```

```
f"{one_half} > {one_third}" # '1/2 > 0.3333333333333333'
```

Custom `__str__` behavior

When making custom classes, we can override `__str__` to define our human readable string representation.

```
class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "Lamb named " + self.name
```

```
lil = Lamb("Lil lamb")

str(lil)

print(lil)
```

__repr__

The `__repr__` method returns a string that would evaluate to an object with the same values.

```
from fractions import Fraction

one_half = Fraction(1, 2)
Fraction.__repr__(one_half)           # 'Fraction(1, 2)'
```

If implemented correctly, calling `eval()` on the result should return back that same-valued object.

```
another_half = eval(Fraction.__repr__(one_half))
```

__repr__ usage

The `__repr__` method is used multiple places by Python: when `repr(object)` is called and when displaying an object in an interactive Python session.

```
from fractions import Fraction
```

```
one_third = 1/3
```

```
one_half = Fraction(1, 2)
```

```
one_third
```

```
one_half
```

```
repr(one_third)
```

```
repr(one_half)
```

Custom `__repr__` behavior

When making custom classes, we can override `__repr__` to return a more appropriate Python representation.

```
class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "Lamb named " + self.name

    def __repr__(self):
        return f"Lamb({repr(self.name)})"
```

```
lil = Lamb("Lil lamb")
repr(lil)
lil
```

Special methods

Special methods

Certain names are special because they have built-in behavior. Those method names always start and end with double underscores.

| Name | Behavior |
|------------------------|--|
| <code>__init__</code> | Method invoked automatically when an object is constructed |
| <code>__repr__</code> | Method invoked to display an object as a Python expression |
| <code>__str__</code> | Method invoked to stringify an object |
| <code>__add__</code> | Method invoked to add one object to another |
| <code>__bool__</code> | Method invoked to convert an object to True or False |
| <code>__float__</code> | Method invoked to convert an object to a float (real number) |

See all special method names.

Special method examples

```
zero = 0  
one = 1  
two = 2
```

Syntactic sugar

Dunder equivalent

```
one + two # 3
```

```
one.__add__(two) # 3
```

```
bool(zero) # False
```

```
zero.__bool__() # False
```

```
bool(one) # True
```

```
one.__bool__() # True
```


Adding together custom objects

Consider the following class:

```
from math import gcd

class Rational:
    def __init__(self, numerator, denominator):
        g = gcd(numerator, denominator)
        self.number = numerator // g
        self.denom = denominator // g

    def __str__(self):
        return f"{self.number}/{self.denom}"

    def __repr__(self):
        return f"Rational({self.number}, {self.denom})"
```

Will this work?

```
Rational(1, 2) + Rational(3, 4)
```

Adding together custom objects

Consider the following class:

```
from math import gcd

class Rational:
    def __init__(self, numerator, denominator):
        g = gcd(numerator, denominator)
        self.number = numerator // g
        self.denom = denominator // g

    def __str__(self):
        return f"{self.number}/{self.denom}"

    def __repr__(self):
        return f"Rational({self.number}, {self.denom})"
```

Will this work?

```
Rational(1, 2) + Rational(3, 4)
```

`TypeError: unsupported operand type(s) for +: 'Rational' and 'Rational'`

Implementing dunder methods

We can make instances of custom classes addable by defining the `__add__` method:

```
class Rational:
    def __init__(self, numerator, denominator):
        g = gcd(numerator, denominator)
        self.numer = numerator // g
        self.denom = denominator // g

    def __add__(self, other):

# The rest...
```

Implementing dunder methods

We can make instances of custom classes addable by defining the `__add__` method:

```
class Rational:
    def __init__(self, numerator, denominator):
        g = gcd(numerator, denominator)
        self.numer = numerator // g
        self.denom = denominator // g

    def __add__(self, other):
        new_numer = self.numer * other.denom + other.numer * self.denom
        new_denom = self.denom * other.denom
        return Rational(new_numer, new_denom)

# The rest...
```

Implementing dunder methods

We can make instances of custom classes addable by defining the `__add__` method:

```
class Rational:
    def __init__(self, numerator, denominator):
        g = gcd(numerator, denominator)
        self.numer = numerator // g
        self.denom = denominator // g

    def __add__(self, other):
        new_numer = self.numer * other.denom + other.numer * self.denom
        new_denom = self.denom * other.denom
        return Rational(new_numer, new_denom)

# The rest...
```

Now try...

```
Rational(1, 2) + Rational(3, 4)
```

Polymorphism

Polymorphic functions

Polymorphic function: A function that applies to many (poly) different forms (morph) of data

`str` and `repr` are both polymorphic; they apply to any object

`repr` invokes a zero-argument method `__repr__` on its argument:

```
one_half = Rational(1, 2)
one_half.__repr__() # 'Rational(1, 2)'
```

`str` invokes a zero-argument method `__str__` on its argument:

```
one_half = Rational(1, 2)
one_half.__str__() # '1/2'
```

Implementing repr and str

The behavior of `repr` is slightly more complicated than invoking `__repr__` on its argument:

- An instance attribute called `__repr__` is ignored! Only class attributes are found
- Poll: How could we implement this behavior?

Implementing repr and str

The behavior of `repr` is slightly more complicated than invoking `__repr__` on its argument:

- An instance attribute called `__repr__` is ignored! Only class attributes are found
- Poll: How could we implement this behavior?

The behavior of `str` is also complicated:

- An instance attribute called `__str__` is ignored
- If no `__str__` attribute is found, uses `repr` string
- (By the way, `str` is a class, not a function)
- Demo: How would we implement this behavior?

Generic functions

A **generic function** can apply to arguments of different types.

```
def sum_two(a, b):  
    return a + b
```

What could `a` and `b` be?

The function `sum_two` is **generic** in the type of `a` and `b`.

Generic functions

A **generic function** can apply to arguments of different types.

```
def sum_two(a, b):  
    return a + b
```

What could `a` and `b` be? Anything summable!

The function `sum_two` is **generic** in the type of `a` and `b`.

Generic function #2

```
def sum_em(items, initial_value):  
    """Returns the sum of ITEMS,  
    starting with a value of INITIAL_VALUE."""  
    sum = initial_value  
    for item in items:  
        sum += item  
    return sum
```

What could `items` be?

What could `initial_value` be?

The function `sum_em` is **generic** in the type of `items` and the type of `initial_value`.

Generic function #2

```
def sum_em(items, initial_value):  
    """Returns the sum of ITEMS,  
    starting with a value of INITIAL_VALUE."""  
    sum = initial_value  
    for item in items:  
        sum += item  
    return sum
```

What could `items` be? Any iterable with summable values.

What could `initial_value` be?

The function `sum_em` is **generic** in the type of `items` and the type of `initial_value`.

Generic function #2

```
def sum_em(items, initial_value):  
    """Returns the sum of ITEMS,  
    starting with a value of INITIAL_VALUE."""  
    sum = initial_value  
    for item in items:  
        sum += item  
    return sum
```

What could `items` be? Any iterable with summable values.

What could `initial_value` be? Any value that can be summed with the values in iterable.

The function `sum_em` is **generic** in the type of `items` and the type of `initial_value`.

Type dispatching

Another way to make generic functions is to select a behavior based on the type of the argument.

```
def is_valid_month(month):  
    if isinstance(month, str) and len(str) == 1:  
        month =  
    if isinstance(month, int):  
        return month >= 1 and month <= 12  
    elif isinstance(month, str):  
        return month in ["January", "February", "March", "April",  
                        "May", "June", "July", "August", "September",  
                        "October", "November", "December"]  
    return false
```

What could `month` be?

The function `is_valid_month` is **generic** in the type of `month`.

Type dispatching

Another way to make generic functions is to select a behavior based on the type of the argument.

```
def is_valid_month(month):  
    if isinstance(month, str) and len(str) == 1:  
        month =  
    if isinstance(month, int):  
        return month >= 1 and month <= 12  
    elif isinstance(month, str):  
        return month in ["January", "February", "March", "April",  
                          "May", "June", "July", "August", "September",  
                          "October", "November", "December"]  
  
    return false
```

What could `month` be? Either an int or string.

The function `is_valid_month` is **generic** in the type of `month`.

Type coercion

Another way to make generic functions is to coerce an argument into the desired type.

```
def sum_numbers(nums):  
    """Returns the sum of NUMS"""  
    sum = Rational(0, 0)  
    for num in nums:  
        if isinstance(num, int):  
            num = Rational(num, 1)  
        sum += num  
    return sum
```

What could `nums` be?

The function `sum_numbers` is **generic** in the type of `nums`.

Type coercion

Another way to make generic functions is to coerce an argument into the desired type.

```
def sum_numbers(nums):  
    """Returns the sum of NUMS"""  
    sum = Rational(0, 0)  
    for num in nums:  
        if isinstance(num, int):  
            num = Rational(num, 1)  
        sum += num  
    return sum
```

What could `nums` be? Any iterable with ints or Rationals.

The function `sum_numbers` is **generic** in the type of `nums`.

Python Project of The Day!

ASCIIfy

ASCIIfy: A Python script to turn an image into an ASCII string, using the [Python pillow library](#).

[illegible]