Generics

Today we will cover..

- String formatting
- Generics/Duck typing
- String joinings

String formatting

Old string formatting methods

Back when I was a wee lass, this was all we had:

```
greeting = "Ahoy"
noun = "Boat"

print(greeting + ", " + noun + "yMc" + noun + "Face")

print("%s, %syMc%sFace" % (greeting, noun, noun))
```

Old string formatting methods

Back when I was a wee lass, this was all we had:

```
greeting = "Ahoy"
noun = "Boat"

print(greeting + ", " + noun + "yMc" + noun + "Face")

print("%s, %syMc%sFace" % (greeting, noun, noun))
```

Python 2.6 introduced `str.format()`:

```
print("{}, {}yMc{}Face".format(greeting, noun, noun))

print("{0}, {1}yMc{1}Face".format(greeting, noun))

print("{greeting}, {noun}yMc{noun}Face".format(
    greeting=greeting, noun=noun))
```

f strings

Available since Python 3.5, **f strings** (formatted string literals) are the new recommended way to format strings.

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

```
greeting = "Ahoy"
noun = "Boat"

print(f"{greeting}, {noun}yMc{noun}Face")
```

Any valid Python expression!

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

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

f strings + Link/Tree

Using our standard 61A definitions, what will this show?

```
ll = Link("A", Link("B", Link("C")))
t = Tree(1, [Tree(2), Tree(3)])

print(f"{ll} and\n {t}")
```

f strings + Link/Tree

Using our standard 61A definitions, what will this show?

```
ll = Link("A", Link("B", Link("C")))
t = Tree(1, [Tree(2), Tree(3)])

print(f"{ll} and\n {t}")
```

It shows the result of calling `__str__` on each object:

```
<A B C> and
    1
     2
     3
```

Generics

A generic function

```
def map_em(items, func):
    """Returns a list with FUNC applied to each item in ITEMS."""
    mapped = []
    for item in items:
        mapped.append(func(item))
    return mapped
```

What could `items` be?

The function `map_em` is **generic** in the type of `items` .

A generic function

```
def map_em(items, func):
    """Returns a list with FUNC applied to each item in ITEMS."""
    mapped = []
    for item in items:
        mapped.append(func(item))
    return mapped
```

What could `items` be? Anything iterable!

The function `map_em` is **generic** in the type of `items` .

What makes something iterable?

The object must have an `__iter__` method that returns an iterator.

Built-in iterables:
`list`, `tuple`, `dict`, `str`, `set`

Built-in functions that return iterables:
`list()`, `tuple()`, `sorted()`

Built-in functions that return iterators:
`reversed()`, `zip()`, `map()`, `filter()`

Duck typing

The ability to use any type of object in a function based on its behavior (versus its type) is known as **duck typing**.

The duck test: 🐥

“The parameter to this function must be a duck. If it looks like a duck and quacks like a duck, then we’ll say it IS a duck!”

Which parameters pass the duck test?

```
map_em([1, 2, 3, 4], lambda n: n * 2)
map_em(("A", "B", "C", "D"), lambda l: l.lower())
map_em({"CA": "poppy", "OR": "grape"}, lambda k: k[0])
map_em([(34, -144), (37, -122)], lambda ll: ll[0])
map_em("Superkalifragilous", lambda s: s.upper())
map_em(Link(1, Link(2, Link(3))), lambda n: n * 3)
```

Why does Link fail?

🚫TypeError: 'Link' object is not iterable

Our 61A standard definition of Link:

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Our object supports initialization and string representations, but not iteration.

Custom iterable

We can make our own objects iterable by defining `__iter__`.

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __iter__(self):
        current = self
        while current is not Link.empty:
            yield current.first
            current = current.rest

    # The rest...
```

Will it duck?

Given the addition of `__iter__`, can we now pass an instance of `Link` into `map_em`?

```
def map_em(items, func):
    """Returns a list with FUNC applied to each item in ITEMS."""
    mapped = []
    for item in items:
        mapped.append(func(item))
    return mapped
```

```
mapped1 = map_em(Link(1, Link(2, Link(3))), lambda n: n * 3)
```

What about an empty linked list?

```
mapped2 = map_em(Link.empty, lambda n: n * 3)
```

Another generic function: sum_em

```
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`.

Another generic function: sum_em

```
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.

What could `initial_value` be?

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

Another generic function: sum_em

```
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.

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`.

Duck typing with sum_em

The duck test: 🐥

“The parameter to this function must be a duck. If it looks like a duck and quacks like a duck, then we’ll say it IS a duck!”

Which parameters pass the duck test?

```
sum_em([1, 2, 3, 4], 0)
sum_em(("H", "E", "L", "L", "O"), "")
sum_em({"CA": "poppy", "OR": "grape"}, "")
sum_em([(10, 20), (30, 40)], (0, 1))
sum_em("Superkalifragilous", "Oh")
sum_em(Link(1, Link(2, Link(3))), 0)
```

Adding custom objects

Consider the following class:

```
from math import gcd

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

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

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

Will it duck?

```
sum_em([Rational(1, 2), Rational(3, 4), Rational(2, 3)],
    Rational(0, 1))
```

Adding custom objects

Consider the following class:

```
from math import gcd

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

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

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

Will it duck?

```
sum_em([Rational(1, 2), Rational(3, 4), Rational(2, 3)],
    Rational(0, 1))
```

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

Custom addable

We can make custom objects 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...
```

P.S. We could also define `__iadd__` to specifically override `+=`.

Custom addable

We can make custom objects 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...
```

P.S. We could also define `__iadd__` to specifically override `+=`.

Will it duck?

Given the addition of `__add__`, can we now pass `Rational` objects into `sum_em`?

```
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
```

```
sum_em([Rational(1, 2), Rational(3, 4), Rational(2, 3)],
    Rational(0, 1))
```

Generic method names

Python has many ways to make custom objects work generically with its syntax. For example:

Method	Implements
`__getitem__(S, k)`	`S[k]`
`__setitem__(S, k, v)`	`S[k] = v`
`__len__(S)`	`len(s)`
`__setattr__(obj, "n", v)`	`x.n = v`
`__delattr__(obj, "n")`	`del x.n`
`__sub__(S, x)`	`S - x`
`__mul__(S, x)`	`S * x`
`__eq__(obj, x)`	`obj == x`
`__lt__(obj, x)`	`obj < x`

...

String joining

Joining strings

The method `str.join(iterable)` returns a string which is the concatenation of the strings in an iterable.

```
names = ["Gray", "Fox"]
print("".join(names))

address_parts = ["123 Pining St", "Nibbsville", "OH"]
print(",".join(address_parts))

poem_lines = ["Forgive me", "they were delicious", "so sweet", "and
print("\n".join(poem_lines))
```

Python documentation: str.join

Joining strings from Link

Using our standard 61A definition, what will this do?

```
ll = Link("A", Link("B", Link("C")))
letters = "->".join(ll)
```

Joining strings from Link

Using our standard 61A definition, what will this do?

```
ll = Link("A", Link("B", Link("C")))
letters = "->".join(ll)
```

🚫 TypeError: 'Link' object is not iterable

What if we use the definition from earlier with `__iter__`?