

The result of calling `repr` on a value is what Python displays in an interactive session

The result of calling `str` on a value is what Python prints using the `print` function

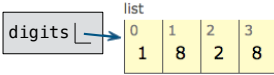
```
>>> today = datetime.date(2019, 10, 13)
>>> repr(today) # or today.__repr__()
'datetime.date(2019, 10, 13)'
>>> str(today) # or today.__str__()
'2019-10-13'
```

The result of evaluating an f-string literal contains the str string of the value of each sub-expression.

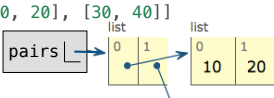
```
>>> f'pi starts with {pi}...'
'pi starts with 3.141592653589793...'
>>> print(f'pi starts with {pi}...')
pi starts with 3.141592653589793...
```

Lists:

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8
```



```
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```



Executing a `for` statement:

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header `<expression>`, which must yield an iterable value (a list, tuple, iterator, etc.)
2. For each element in that sequence, in order:
 - A. Bind `<name>` to that element in the current frame
 - B. Execute the `<suite>`

Unpacking in a `for` statement:

```
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]
>>> same_count = 0
```

A name for each element in a fixed-length sequence

```
>>> for x, y in pairs:
...     if x == y:
...         same_count = same_count + 1
>>> same_count
2
```

..., -3, -2, -1, 0, 1, 2, 3, 4, ...

range(-2, 2)

Length: ending value - starting value

Element selection: starting value + index

```
>>> list(range(-2, 2))
[-2, -1, 0, 1]
```

```
>>> list(range(4))
[0, 1, 2, 3]
```

Membership:

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Identity:

`<exp0> is <exp1>` evaluates to `True` if both `<exp0>` and `<exp1>` evaluate to the same object

Equality:

`<exp0> == <exp1>` evaluates to `True` if both `<exp0>` and `<exp1>` evaluate to equal values

Identical objects are always equal values

```
iter(iterable):
    Return an iterator over the elements
    of an iterable value
next(iterator):
    Return the next element of an iterator
```

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> next(t)
5
```

```
>>> d = {'one': 1, 'two': 2}
>>> k = iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
```

A **generator function** is a function that **yields** values instead of **returning**.

```
>>> def plus_minus(x):
...     yield x
...     yield -x
>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
```

```
>>> def a_then_b(a, b):
...     yield from a
...     yield from b
>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]
```

List comprehensions:

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

Short version: `[<map exp> for <name> in <iter exp>]`

A combined expression that evaluates to a list using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty *result list* that is the value of the expression
3. For each element in the iterable value of `<iter exp>`:
 - A. Bind `<name>` to that element in the new frame from step 1
 - B. If `<filter exp>` evaluates to a true value, then add the value of `<map exp>` to the result list

Dictionaries:

```
words = {
    "más": "more",
    "otro": "other",
    "agua": "water"
}
```

```
>>> len(words)
3
>>> "agua" in words
True
>>> words["otro"]
'other'
>>> words["pavo"]
KeyError
>>> words.get("pavo", "🐔")
'🐔'
```

Dictionary comprehensions:

```
{key: value for <name> in <iter exp>}
>>> {x: x*x for x in range(3,6)}
{3: 9, 4: 16, 5: 25}
```

```
>>> [word for word in words]
['más', 'otro', 'agua']
>>> [words[word] for word in words]
['more', 'other', 'water']
>>> words["oruguita"] = 'caterpillar'
>>> words["oruguita"]
'caterpillar'
>>> words["oruguita"] += '🐛'
>>> words["oruguita"]
'caterpillar🐛'
```

Functions that aggregate iterable arguments

```
• sum(iterable[, start]) -> value      sum of all values
• max(iterable[, key=func]) -> value  largest value
  max(a, b, c, ..., key=func) -> value
• min(iterable[, key=func]) -> value  smallest value
  min(a, b, c, ..., key=func) -> value
• all(iterable) -> bool               whether all are true
• any(iterable) -> bool               whether any is true
```

Many built-in Python sequence operations return iterators that compute results lazily

To view the contents of an iterator, place the resulting elements into a container

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)
```

```
>>> cascade(123)
123
12
1
```

```
def virfib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return virfib(n-2) + virfib(n-1)
```

```
n: 0, 1, 2, 3, 4, 5, 6, 7, 8,
virfib(n): 0, 1, 1, 2, 3, 5, 8, 13, 21,
```

Exponential growth. E.g., recursive `fib`

Incrementing n multiplies *time* by a constant

$\Theta(b^n)$ $O(b^n)$

Quadratic growth. E.g., `overlap`

Incrementing n increases *time* by n times a constant

$\Theta(n^2)$ $O(n^2)$

Linear growth. E.g., `slow exp`

Incrementing n increases *time* by a constant

$\Theta(n)$ $O(n)$

Logarithmic growth. E.g., `exp_fast`

Doubling n only increments *time* by a constant

$\Theta(\log n)$ $O(\log n)$

Constant growth. Increasing n doesn't affect *time*

$\Theta(1)$ $O(1)$



List mutation:

```
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a == b
True
>>> a
[10, 20]
>>> b
[10, 20]
>>> a == b
False
```

You can **copy** a list by calling the list constructor or slicing the list from the beginning to the end.

```
>>> a = [10, 20, 30]
>>> list(a)
[10, 20, 30]
>>> a[:]
[10, 20, 30]
```

Tuples:

```
>>> empty = ()
>>> len(empty)
0
>>> conditions = ('rain', 'shine')
>>> conditions[0]
'rain'
>>> conditions[0] = 'fog'
Error
```

```
>>> all([False, True])
False
>>> all([1])
True
>>> sum([1, 2])
3
>>> sum([1, 2], 3)
6
>>> sum([1])
0
>>> sum([1], [2], [])
[1, 2]
```

```
>>> any([False, True])
True
>>> any([1])
False
>>> max([1, 2])
2
>>> max([1, 2])
2
>>> max([1, -2], key=abs)
-2
```

List methods:

```
>>> suits = ['coin', 'string', 'myriad']
>>> suits.pop()
'myriad'
>>> suits.remove('string')
```

Remove and return the last element
Removes first matching value

```
>>> suits.append('cup')
>>> suits.extend(['sword', 'club'])
>>> suits[2] = 'spade'
>>> suits
['coin', 'cup', 'spade', 'club']
>>> suits[0:2] = ['diamond']
>>> suits
['diamond', 'spade', 'club']
>>> suits.insert(0, 'heart')
>>> suits
['heart', 'diamond', 'spade', 'club']
```

Add all values
Replace a slice with values
Add an element at an index

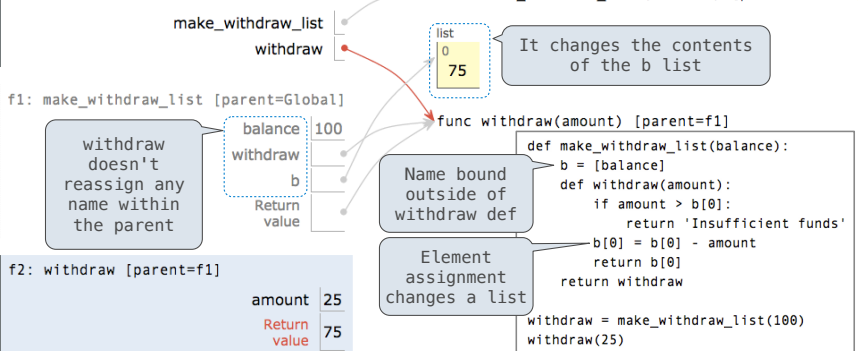
False values:

```
• Zero
• False
• None
• An empty string, list, dict, tuple
```

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool('')
False
>>> bool('0')
True
>>> bool([])
False
>>> bool({})
False
>>> bool({})
False
>>> bool(lambda x: 0)
True
```



Global frame

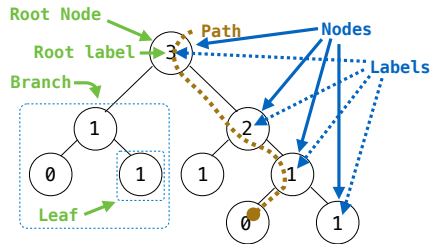


Recursive description:

- A tree has a root label and a list of branches
- Each branch is a tree
- A tree with zero branches is called a leaf

Relative description:

- Each location is a node
- Each node has a label
- One node can be the parent/child of another



```
def tree(label, branches=[]):
```

```
    for branch in branches:
```

```
        assert is_tree(branch)
```

```
    return [label] + list(branches)
```

```
def label(tree):
```

```
    return tree[0]
```

```
def branches(tree):
```

```
    return tree[1:]
```

```
def is_tree(tree):
```

```
    if type(tree) != list or len(tree) < 1:
```

```
        return False
```

```
    for branch in branches(tree):
```

```
        if not is_tree(branch):
```

```
            return False
```

```
    return True
```

```
def is_leaf(tree):
```

```
    return not branches(tree)
```

```
def leaves(t):
```

```
    """The leaf values in t.
```

```
    >>> leaves(fib_tree(5))
```

```
    [1, 0, 1, 0, 1, 1, 0, 1]
```

```
    """
```

```
    if is_leaf(t):
```

```
        return [label(t)]
```

```
    else:
```

```
        return sum([leaves(b) for b in branches(t)], [])
```

```
class Tree:
```

```
    def __init__(self, label, branches=[]):
```

```
        self.label = label
```

```
        for branch in branches:
```

```
            assert isinstance(branch, Tree)
```

```
        self.branches = list(branches)
```

```
    def is_leaf(self):
```

```
        return not self.branches
```

```
>>> b = Tree(2, [Tree(3)])
```

```
>>> t = Tree(1, [b, Tree(4)])
```

```
>>> t
```

```
Tree(1, [Tree(2, [Tree(3)]), Tree(4)])
```

```
>>> print(t)
```

```
1
2 3
4
```

```
class Link:
```

```
    empty = ()
```

```
    def __init__(self, first, rest=empty):
```

```
        self.first = first
```

```
        self.rest = rest
```

```
    def __repr__(self):
```

```
        if self.rest:
```

```
            rest = ', ' + repr(self.rest)
```

```
        else:
```

```
            rest = ''
```

```
        return 'Link(' + repr(self.first) + rest + ')'
```

```
    def __str__(self):
```

```
        string = '<'
```

```
        while self.rest is not Link.empty:
```

```
            string += str(self.first) + ' '
```

```
            self = self.rest
```

```
        return string + str(self.first) + '>'
```

Anatomy of a recursive function:

- The **def statement header** is like any function
- Conditional statements check for **base cases**
- Base cases are evaluated **without recursive calls**
- Recursive cases are evaluated **with recursive calls**

```
def sum_digits(n):
```

```
    """Sum the digits of positive integer n."""
```

```
    if n < 10:
```

```
        return n
```

```
    else:
```

```
        all_but_last, last = n // 10, n % 10
```

```
        return sum_digits(all_but_last) + last
```

Recursive decomposition: finding simpler instances of a problem.

```
def count_partitions(n, m):
```

```
    if n == 0:
```

```
        return 1
```

```
    elif n < 0:
```

```
        return 0
```

```
    elif m == 0:
```

```
        return 0
```

```
    else:
```

```
        with_m = count_partitions(n-m, m)
```

```
        without_m = count_partitions(n, m-1)
```

```
        return with_m + without_m
```

```
Tree recursion often involves exploring different choices.
```

Python object system:

Idea: All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

A new instance is created by calling a class

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

An account instance

```
balance: 0 holder: 'Jim'
```

When a class is called:

1. A new instance of that class is created:

2. The `__init__` method of the class is called with the new object as its first argument (named `self`), along with any additional arguments provided in the call expression.

```
class Account:
```

```
    def __init__(self, account_holder):
```

```
        self.balance = 0
```

```
        self.holder = account_holder
```

```
    def deposit(self, amount):
```

```
        self.balance = self.balance + amount
```

```
        return self.balance
```

```
    def withdraw(self, amount):
```

```
        if amount > self.balance:
```

```
            return 'Insufficient funds'
```

```
        self.balance = self.balance - amount
```

```
        return self.balance
```

`__init__` is called a constructor

`self` should always be bound to an instance of the Account class or a subclass of Account

Function call: all arguments within parentheses

```
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>
```

Method invocation: One object before the dot and other arguments within parentheses

```
>>> Account.deposit(a, 5)
10
>>> a.deposit(2)
12
```

Call expression

Dot expression

`<expression> . <name>`

The `<expression>` can be any valid Python expression.

The `<name>` must be a simple name.

Evaluates to the value of the attribute looked up by `<name>` in the object that is the value of the `<expression>`.

To evaluate a dot expression:

1. Evaluate the `<expression>` to the left of the dot, which yields the object of the dot expression
2. `<name>` is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
3. If not, `<name>` is looked up in the class, which yields a class attribute value
4. That value is returned unless it is a function, in which case a bound method is returned instead

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

Account class attributes

```
interest: 0.00 0.04 0.05
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

```
balance: 0
holder: 'Jim'
interest: 0.08
```

Instance attributes of tom_account

```
balance: 0
holder: 'Tom'
```

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04
```

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

```
class CheckingAccount(Account):
```

```
    """A bank account that charges for withdrawals."""
```

```
    withdraw_fee = 1
```

```
    interest = 0.01
```

```
    def withdraw(self, amount):
```

```
        return Account.withdraw(self, amount + self.withdraw_fee)
```

```
        or
        return super().withdraw(amount + self.withdraw_fee)
```

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom') # Calls Account.__init__
>>> ch.interest # Found in CheckingAccount
0.01
>>> ch.deposit(20) # Found in Account
20
>>> ch.withdraw(5) # Found in CheckingAccount
14
```