Control
Class outline:

- Side effects
- More function features
- Conditionals
- Booleans
- Iteration
Side effects
The None value

The special value `None` represents nothingness in Python. Any function that doesn't explicitly return a value will return `None`:

```python
def square_it(x):
    x * x
```
The None value

The special value `None` represents nothingness in Python. Any function that doesn't explicitly return a value will return `None`:

```python
def square_it(x):
    x * x
```

When a function returns `None`, the console shows no output at all:

```python
square_it(4)
```
The None value

The special value `None` represents nothingness in Python. Any function that doesn't explicitly return a value will return `None`:

```python
def square_it(x):
    x * x
```

When a function returns `None`, the console shows no output at all:

```python
square_it(4)
```

Attempting to treat the `None` like a number will result in an error:

```python
sixteen = square_it(4)
sum = sixteen + 4  # TypeError!
```
Side effects

A **side effect** is when something happens as a result of calling a function besides just returning a value.

The most common side effect is logging to the console, via the built-in `print()` function.

```
print(-2)
```

Other common side effects: writing to files, drawing graphics on the screen.
Side effects vs. Return values

```python
def square_num1(number):
    return pow(number, 2)

def square_num2(number):
    print(number ** 2)
```

- Which one has a side effect?
- What data type do they each return?
Side effects vs. Return values

```python
def square_num1(number):
    return pow(number, 2)
```

```python
def square_num2(number):
    print(number ** 2)
```

- Which one has a side effect? The second function has a side effect, because it prints to the console.

- What data type do they each return?
Side effects vs. Return values

```python
def square_num1(number):
    return pow(number, 2)

def square_num2(number):
    print(number ** 2)
```

- Which one has a side effect?
  The second function has a side effect, because it prints to the console.

- What data type do they each return?
  The first function returns a number, the second one returns `None`. 
Pure vs. non-pure functions

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure functions</td>
<td>just return values.</td>
</tr>
<tr>
<td>-2</td>
<td>2</td>
</tr>
<tr>
<td>2, 10</td>
<td>1024</td>
</tr>
</tbody>
</table>
# Pure vs. non-pure functions

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure functions</td>
<td>just return values.</td>
</tr>
<tr>
<td>-2</td>
<td>2</td>
</tr>
<tr>
<td>2, 10</td>
<td>1024</td>
</tr>
<tr>
<td>Non-pure functions</td>
<td>have side effects.</td>
</tr>
<tr>
<td>-2</td>
<td>None</td>
</tr>
</tbody>
</table>

Python displays output "-2"
Nested print statements

What will this display?

```python
print(print(1), print(2))
```
Nested print statements

What will this display?

```
print(print(1), print(2))
```

```
print(print(1), print(2))
```
Nested print statements

What will this display?

```python
print(print(1), print(2))
```

```
def print()...
```

```python
print(print(1), print(2))
```
Nested print statements

What will this display?

```
print(print(1), print(2))
```

def print()...

```
print(print(1), print(2))
```

```
def print()...
    print(1)
```
Nested print statements

What will this display?

```
print(print(1), print(2))
```
Nested print statements

What will this display?

```
print(print(1), print(2))
```
Nested print statements

What will this display?

```
print(print(1), print(2))
```

```
def print()...
print(1)
def print()...
```
Nested print statements

What will this display?

```python
print(print(1), print(2))
```

```
def print()...

print(1)
def print()...

Display "1"
```
Nested print statements

What will this display?

```
print(print(1), print(2))
```

```
def print()...
    print(1)
def print()...
    print(2)
    return None
```

Display "1"
Nested print statements

What will this display?

```
print(print(1), print(2))
```

```
def print()...
  print(1)

def print()...
  print(2)
```

Display "1"
Nested print statements

What will this display?

\[
\text{print(print(1), print(2))}
\]
Nested print statements

What will this display?

```
print(print(1), print(2))
```
Nested print statements

What will this display?

```python
print(print(1), print(2))
```
Nested print statements

What will this display?

```
print(print(1), print(2))
```
Nested print statements

What will this display?

```
print(print(1), print(2))
```

```
def print()...
    print(1)

def print()...
    print(2)
```

Display "None None"

Display "1"

Display "2"
More function features
Default arguments

In the function signature, a parameter can specify a **default value**. If that argument isn't passed in, the default value is used instead.

```python
def calculate_dog_age(human_years, multiplier = 7):
    return human_years * multiplier
```

These two lines of code have the same result:

```python
calculate_dog_age(3)
calculate_dog_age(3, 7)
```

Default arguments can be overriding two ways:

```python
calculate_dog_age(3, 6)
calculate_dog_age(3, multiplier=6)
```
Multiple return values

A function can specify multiple return values, separated by commas.

```python
def divide_exact(n, d):
    quotient = n // d
    remainder = n % d
    return quotient, remainder
```

Any code that calls that function must also "unpack it" using commas:

```python
q, r = divide_exact(618, 10)
```
Doctests

Doctests check the input/output of functions.

```python
def divide_exact(n, d):
    
    >>> q, r = divide_exact(2021, 10)
    >>> q
    202
    >>> r
    1
    
    quotient = n // d
    remainder = n % d
    return quotient, remainder
```

See more in Python doctests documentation.
Boolean expressions
Booleans

A **Boolean value** is either **True** or **False** and is used frequently in computer programs.

Google Maps uses a boolean to decide whether to avoid highways in driving directions:

```
avoid_highways = True
```

Twitter uses a boolean to remember where the user allows personalized ads:

```
personalized_ads = False
```
Boolean expressions

An expression can evaluate to a Boolean. Most Boolean expressions use either comparison or logical operators.

An expression with a comparison operator:

```python
passed_class = grade > 65
```

An expression with a logical operator:

```python
wear_jacket = is_raining or is_windy
```
# Comparison operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>True expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>Equality</td>
<td><code>32 == 32, 'a' == 'a'</code></td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Inequality</td>
<td><code>30 != 32, 'a' != 'b'</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td><code>60 &gt; 32</code></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal</td>
<td><code>60 &gt;= 32, 32 &gt;= 32</code></td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td><code>20 &lt; 32</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal</td>
<td><code>20 &lt;= 32, 32 &lt;= 32</code></td>
</tr>
</tbody>
</table>

⚠ Common mistake: Do not confuse `=` (the assignment operator) with `==` (the equality operator).
## Logical operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>True expressions</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>and</strong></td>
<td>$4 &gt; 0 \text{ and } -2 &lt; 0$</td>
<td>Evaluates to <strong>True</strong> if both conditions are true. If one is <strong>False</strong> evaluates to <strong>False</strong>.</td>
</tr>
<tr>
<td><strong>or</strong></td>
<td>$4 &gt; 0 \text{ or } -2 &gt; 0$</td>
<td>Evaluates to <strong>True</strong> if either condition is true. Evaluates to <strong>False</strong> only if both are false.</td>
</tr>
<tr>
<td><strong>not</strong></td>
<td>not $(5 == 0)$</td>
<td>Evaluates to <strong>True</strong> if condition is false; evaluates to <strong>False</strong> if condition is true.</td>
</tr>
</tbody>
</table>
Compound booleans

When combining multiple operators in a single expression, use parentheses to group:

```python
may_have_mobility_issues = (age >= 0 and age < 2) or age > 90
```
Boolean expressions in functions

A function can use a Boolean expression to return a result based on the values of the parameters.

```python
def passed_class(grade):
    return grade > 65

def should_wear_jacket(is_rainy, is_windy):
    return is_rainy or is_windy
```
Exercise

These are un-graded exercises you can do after the lecture to make sure you grok the basics:

- `has_curly_hair()
- `can_be_president()
- `is_safe_to_eat()
- `harvest_time()`
Statements
## Statements

A **statement** is executed by the interpreter to perform an action.

So far we've seen a few...

<table>
<thead>
<tr>
<th>Statement type</th>
<th>Example</th>
</tr>
</thead>
</table>
| Assignment statement    | name = 'sosuke'
                        | greeting = 'ahoy, ' + name                                              |
| Def statement           | def greet(name):
                        |     return 'ahoy, ' + name                                              |
| Return statement        | return 'ahoy, ' + name                                                  |
Compound statements

A **compound statement** contains groups of other statements.

```plaintext
<header>:
    <statement>
    <statement>
    ...

<separating header>:
    <statement>
    <statement>
    ...
```
Compound statements

A **compound statement** contains groups of other statements.

```
<header>:
    <statement>
    <statement>
    ...

<separating header>:
    <statement>
    <statement>
    ...
```
Compound statements

A **compound statement** contains groups of other statements.

```
<header>:
    <statement>  # SUITE
    <statement>
    ...
</header>:

<separating_header>:
    <statement>  # SUITE
    <statement>
    ...
```
Compound statements

A **compound statement** contains groups of other statements.

```
<header>:                               # CLAUSE
  <statement>  # SUITE
  <statement>
  ...

<separating_header>:                   # CLAUSE
  <statement>  # SUITE
  <statement>
  ...
```

The first header determines a statement's type, and the header of each clause controls the suite that follows.
Execution of suites

A **suite** is a sequence of statements.

```xml
<header>:
  <statement>
  <statement>
  ...
</header>

<separating header>:
  <statement>
  <statement>
  ...
</separating header>

Execution rule for a sequence of statements:

- Execute the first statement
- Unless directed otherwise, execute the rest
Execution of suites

A **suite** is a sequence of statements.

```plaintext
<header>:
<statement>  # SUITE
<statement>
...

<separating header>:
<statement>  # SUITE
<statement>
...
```

Execution rule for a sequence of statements:

- Execute the first statement
- Unless directed otherwise, execute the rest
Conditional statements
Conditional statements

A **conditional statement** gives your code a way to execute a different suite of code statements based on whether certain conditions are true or false.

```python
if <condition>:
    <statement>
    <statement>
    ...
```

A simple conditional:

```python
clothing = "shirt"

if temperature < 32:
    clothing = "jacket"
```
Compound conditionals

A conditional can include any number of `elif` statements to check other conditions.

```python
if <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...

Clothing = "shirt"

if temperature < 0:
    clothing = "snowsuit"
elif temperature < 32:
    clothing = "jacket"
```
The else statement

A conditional can include an `else` to specify code to execute if no previous conditions are true.

```python
if <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...
else <condition>:
    <statement>
    ...
```

```python
if temperature < 0:
    clothing = "snowsuit"
elif temperature < 32:
    clothing = "jacket"
else:
    clothing = "shirt"
```
Conditional statements summary

```python
if num < 0:
    sign = "negative"
elif num > 0:
    sign = "positive"
else:
    sign = "neutral"
```

Syntax tips:

- Always start with **if** clause.
- Zero or more **elif** clauses.
- Zero or one **else** clause, always at the end.
Execution of conditional statements

Each clause is considered in order.

- Evaluate the header's expression.
- If it's true, execute the suite of statements underneath and skip the remaining clauses.
- Otherwise, continue to the next clause.

```python
num = 5
if num < 0:
    sign = "negative"
elif num > 0:
    sign = "positive"
else:
    sign = "neutral"
```

Global frame

<table>
<thead>
<tr>
<th>num</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>&quot;positive&quot;</td>
</tr>
</tbody>
</table>

View in PythonTutor
Conditionals in functions

It's common for a conditional to be based on the value of the parameters to a function.

```python
def get_number_sign(num):
    if num < 0:
        sign = "negative"
    elif num > 0:
        sign = "positive"
    else:
        sign = "neutral"
    return sign

get_number_sign(50)  # "positive"
get_number_sign(-1)  # "negative"
get_number_sign(0)   # "neutral"```
Returns inside conditionals

A branch of a conditional can end in a return, which exits the function entirely.

def get_number_sign(num):
    if num < 0:
        return "negative"
    elif num > 0:
        return "positive"
    else:
        return "neutral"

def get_number_sign(num):
    if num < 0:
        return "negative"
    elif num > 0:
        return "positive"
    else:
        return "neutral"
Exercise

These are un-graded exercises you can do after the lecture to make sure you grok the basics:

- greater_num
- hello_world
- assign_grade
While loops
While loops

The while loop syntax:

```
while <condition>:
    <statement>
    <statement>
```

As long as the condition is true, the statements below it are executed.

```
multiplier = 1
while multiplier <= 5:
    print(9 * multiplier)
    multiplier += 1
```

The code is significantly shorter, and it can easily be extended to loop for more or less iterations.
Using a counter variable

It's common to use a **counter variable** whose job is keeping track of the number of iterations.

```python
total = 0
counter = 0
while counter < 5:
    total += pow(2, 1)
    counter += 1
```

The counter variable may also be involved in the loop computation:

```python
total = 0
counter = 0
while counter < 5:
    total += pow(2, counter)
    counter += 1
```
Beware infinite loops

Uh oh..

counter = 1
while counter < 5:
    total += pow(2, counter)

What one line of code would fix this?
Beware infinite loops

Uh oh..

counter = 1
while counter < 5:
    total += pow(2, counter)

What one line of code would fix this?
counter += 1
Beware infinite loops

Uh oh..

counter = 1
while counter < 5:
    total += pow(2, counter)

What one line of code would fix this?
counter += 1

counter = 6
while counter > 5:
    total += pow(2, counter)
    counter += 1

How do we save this code?
Beware infinite loops

Uh oh..

```python
counter = 1
while counter < 5:
    total += pow(2, counter)
```

What one line of code would fix this?
```python
counter += 1
```

```python
counter = 6
while counter > 5:
    total += pow(2, counter)
    counter += 1
```

How do we save this code?
Intentions are unclear! Change the initial value and condition?
Execution of loops

1. Evaluate the header’s Boolean expression.
2. If it is a true value, execute the suite of statements, then return to step 1.

```
1    sum = 0
2    counter = 0
3    while counter < 10:
4        sum += pow(counter, 2)
5    counter += 1
```

Global frame

- sum: 0
- counter: 1

View in PythonTutor
Loops in functions

A loop in a function will commonly use a parameter to determine some aspect of its repetition.

```python
def sum_up_squares(start, end):
    counter = start
    total = 0
    while counter <= end:
        total += pow(counter, 2)
        counter += 1
    return total

sum_up_squares(1, 5)
```
The break statement

To prematurely exit a loop, use the `break` statement:

counter = 100
while counter < 200:
    if counter % 7 == 0:
        first_multiple = counter
        break
    counter += 1

View in PythonTutor
Looping while true

If you are brave, you can write while loops like this:

counter = 100
while True:
    if counter % 62 == 0:
        first_multiple = counter
        break
    counter += 1

⚠ Be very sure that you're not coding an infinite loop!

Don't trust me? Ask Twitter!
Exercise

These are un-graded exercises you can do after the lecture to make sure you grok the basics:

- `count_evens()`
- `count_multiples()`
- `sum_multiples()`
- `product_of_numbers()`
Example: Prime factors

A **prime number** is an integer greater than 1 whose only factors are 1 and the number itself (e.g., 3, 5, 7, 11).

```python
def is_prime(n):
    """Return True iff N is prime.""
    return n > 1 and smallest_factor(n) == n

def smallest_factor(n):
    """Returns the smallest value k>1 that evenly divides N.""

    ???

def print_factors(n):
    """Print the prime factors of N.""

    ???
```

Let's implement them together.