CSC 221: Introduction to Programming

Fall 2018

Functions & conditionals

- user-defined functions, computational abstraction
  - parameters, local variables
  - return statements
  - function doc strings using """
- conditionals
  - if, if-else, boolean expressions
  - cascading if-else, elif

User-defined functions

in addition to built-in and standard module functions, you can organize your code into functions

- recall the population growth code

population.py - /Users/davereed/Documents/Classes/CSC221/CSC221.F18/Code/populat...

```
# population.py     Dave Reed     9/9/18
#
# Estimate the U.S. population in some number of years in the future.
#####################################################################

population = 328531956

years_in_future = int(input("How many years in the future? "))
seconds_in_future = years_in_future * 365 * 24 * 60 * 60

births = seconds_in_future // 8
deaths = seconds_in_future // 12
naturalized = seconds_in_future // 28

new_population = population + births - deaths + naturalized

print("After", years_in_future, "years, the population would be", \
      new_population)
```

Ln: 8 Col: 0

as is, you have to rerun the module each time you want to calculate a new growth estimate

better solution – package the code into a function, then can call that function repeatedly

Simple functions

in its simplest form, a Python function is a sequence of statements grouped together into a block

- general form:

```
def FUNC_NAME():
    SEQUENCE_OF_STATEMENTS
```

population.py - /Users/davereed/Documents/Classes/CSC221/CSC221.F18/Code/population.p...

```
# population.py     Dave Reed     9/9/18
#
# Estimate the U.S. population in some number of years in the future.
######################################################################

def estimate():
    population = 328531956

    years_in_future = int(input("How many years in the future? "))
    seconds_in_future = years_in_future * 365 * 24 * 60 * 60

    births = seconds_in_future // 8
    deaths = seconds_in_future // 12
    naturalized = seconds_in_future // 28

    new_population = population + births - deaths + naturalized

    print("After", years_in_future, "years, the population would be", \
          new_population)
```

Ln: 20 Col: 0

the statements that make up the function block must be indented (consistently)

you can have blank lines inside the function for readability

consecutive blank lines will end the function

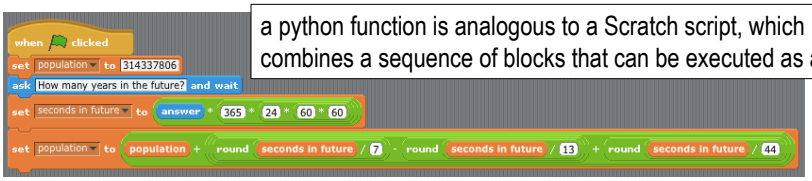
3

Advantages of functions

functions encapsulate a series of statements under a name

- once defined, a function can be called over and over without rerunning the module

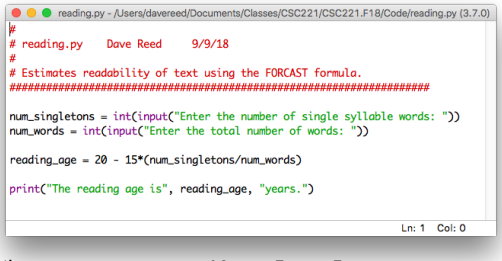
```
>>> estimate()
How many years in the future? 10
After 10 years, the population would be 352934813
>>>
>>> estimate()
How many years in the future? 20
After 20 years, the population would be 377337670
```



a python function is analogous to a Scratch script, which combines a sequence of blocks that can be executed as a unit

4

Exercise: FORCAST



convert your FORCAST reading level code into a function

- encapsulate the code under a function name, e.g., `reading_level`
- test your function by running the module and then doing multiple reading level calculations

simple functions like these are units of *computational abstraction*

- the function encapsulates a sequence of computations into a single unit
- once you know the name you no longer need to remember how it works (i.e., you abstract away the details and focus on the overall effect)

but they don't resemble math functions very closely

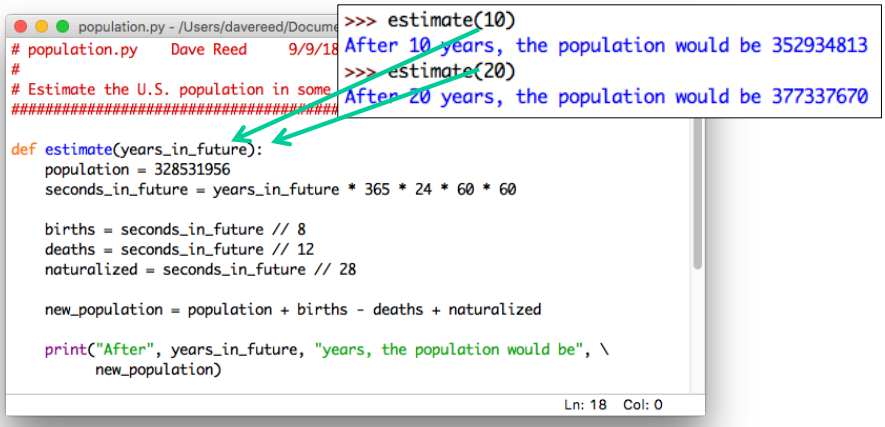
- no inputs & no explicit output

5

Functions with inputs

to generalize a function so that it can take inputs, must add *parameters*

- a parameter is a variable that appears in the heading of a function definition (inside the parentheses)
- when the function is called, you must provide an input value for each parameter
- these input values are automatically assigned to the parameter variables



6

Exercise: parameterizing FORCAST

modify your `reading+level` function to have two parameters

```
def reading_level(single_words, total_words):
```

- note that the input values match up with parameters in the order provided

```
>>> reading_level(104, 380)
```

prompts vs. inputs?

- if your code requires many different input values, then having a function that prompts for each value is often simpler (prompts remind the user of what to enter)
- if there are few inputs, then often simpler & quicker to provide them as inputs
  - ✓ avoids the prompts and allows the user to enter the values directly
  - ✓ allows you to determine the inputs some other way (e.g., another function to calculate the input values)

7

Functions that return values

consider functions that convert distances in Metric & Imperial

- a `return` statement specifies the output value for a function

```
def feet_to_meters(num_feet):
    num_meters = num_feet * 0.3048
    return num_meters
```

```
>>> feet_to_meters(1)
0.3048
>>> feet_to_meters(10)
3.048
>>> feet_to_meters(5280)
1609.344
```

the reverse conversion could be similarly defined using the 0.3048 factor

- or, could make use of the existing `feetToMeters` function

```
def meters_to_feet(num_meters):
    num_feet = num_meters / 0.3048
    return num_feet
```

```
def meters_to_feet(num_meters):
    num_feet = num_meters / feet_to_meters(1)
    return num_feet
```

8

Print vs. return

when the results are complex, printing is more readable

- the population results involve two numbers and description text

```
>>> estimate(1000)
After 1000 years, the population would be 2768817670
```

- much easier to understand than

```
>>> estimate(1000)
2768817670
```

however, a function that prints its results is limited

- the user can view the displayed results, but it cannot be used by other code
- if returned, the result can be used in other computations

```
>>> estimate(1000) *0.90
2491935903.0
```

if the result of a computation is potentially useful for some other computation, then return that result

9

Exercise: years vs. seconds

define a function that converts a number of years into the corresponding number of seconds

```
>>> years_to_seconds(1)
31536000
>>> years_to_seconds(20)
630720000
```

similarly, define a function that does the opposite conversion

```
>>> seconds_to_years(31536000)
1.0
>>> seconds_to_years(1000000)
0.031709791983764585
```

10

Function doc strings

it is considered good programming practice to put a multiline comment at the beginning of every function definition

- it should briefly document the purpose of the function

```
def feet_to_meters(num_feet):
    """Converts from feet to meters."""
    num_meters = num_feet * 0.3048
    return num_meters
```

- in addition to making it easier to read the code in the editor, the doc string can be viewed in the interpreter shell using the built-in `help` function

```
>>> help(feet_to_meters)
Help on function feet_to_meters in module __main__:

feet_to_meters(num_feet)
    Converts from feet to meters.
```

11

Simulating dice rolls

Consider the following function for simulating the roll of a die

```
import random

def roll_die():
    """Simulates rolling a die."""
    roll = random.randint(1, 6)
    print("You rolled", roll)
```

- what is good about this code? what is bad?
- suppose we wanted to generalize the function so that it works for an N-sided die
- suppose we wanted to get the sum of two die rolls

12

Example: wind chill

suppose we wanted to calculate the wind chill using this formula

$$\text{wind chill} = 35.74 + 0.6215*\text{temp} + (0.4275*\text{temp} - 35.75)*\text{wind}^{0.16}$$

```
def wind_chill(temp, wind):
    """Calculates the wind chill factor."""
    return 35.74 + 0.6215*temp + (0.4275*temp - 35.75)*(wind**0.16)
```

would the following return statement suffice?

```
return 35.74 + 0.6215*temp + 0.4275*temp-35.75 * wind**0.16
```

13

Complex expressions

Python has rules that dictate the order in which evaluation takes place

- ** has higher precedence, followed by * and /, then + and –
- meaning that you evaluate the part involving ** first, then * or /, then + or –

```
1 + 2 * 3 → 1 + (2 * 3) → 1 + 6 → 7
2 ** 10 – 1 → (2**10) – 1 → 1024 – 1 → 1023
```

- if more than one operator, ** evaluates right-to-left, all others evaluate left-to-right

```
8 / 4 / 2 → (8 / 4) / 2 → 2 / 2 → 1
2 ** 3 ** 2 → 2 ** (3 ** 2) → 2 ** 9 → 512
```

GOOD ADVICE: don't rely on these (sometimes tricky) rules

- place parentheses around sub-expressions to force the desired order

```
35.74 + 0.6215*temp + (0.4275*temp - 35.75)*(wind**0.16)
```

14

Conditional execution

so far, all of the statements in methods have executed *unconditionally*

- when a method is called, the statements in the body are executed in sequence
- different parameter values may produce different results, but the steps are the same

many applications require *conditional execution*

- different parameter values may cause different statements to be executed

for example, consider the `windChill` formula

- the formula only applies when wind speed > 3 mph
- if wind speed is ≤ 3 mph, wind chill is the same as the temperature

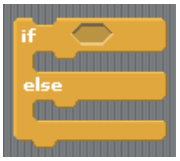
$$\text{wind chill} = \begin{cases} \text{temp} & , \text{ if wind} \leq 3 \\ 35.74 + 0.6215 * \text{temp} + (0.4275 * \text{temp} - 35.75) * \text{wind}^{0.16}, & \text{otherwise} \end{cases}$$

15

If statements

in Python, an *if statement* allows for conditional execution

- i.e., can choose between 2 alternatives to execute




```
if TEST_CONDITION:
    STATEMENTS_TO_EXECUTE_IF_TEST_IS_TRUE
else:
    STATEMENTS_TO_EXECUTE_IF_TEST_IS_FALSE
```

```
def wind_chill(temp, wind):
    """Calculates the wind chill factor."""
    if wind <= 3:
        return temp
    else:
        return 35.74 + 0.6215*temp + \
               (0.4275*temp - 35.75)*(wind**0.16)
```

if the test is true (wind ≤ 3), then this statement is executed

otherwise (wind > 3), then this statement is executed

16

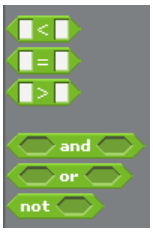
Boolean operators

standard relational operators are provided for the if test

`<`	less than	`>`	greater than
`<=`	less than or equal to	`>=`	greater than or equal to
`==`	equal to	`!=`	not equal to

`and` `or` `not`

a comparison using a relational operator is known as a *Boolean expression*, since it evaluates to a *Boolean* (`True` or `False`) value



EXERCISE:

```
def flip_coin():
    """Simulates flipping a coin."""
    return random.choice(["heads", "tails"])
```

- reimplement using `randint` and an if-else statement
  1. generate a random integer in range [1, 2]
  2. if the number is 1, then return "heads
  3. else, return "tails"

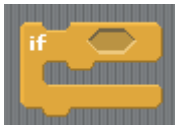
17

If statements (cont.)

you are not required to have an else case to an if statement

- if no else case exists and the test evaluates to false, nothing is done

```
def scale(grade, amount):
    """Returns the grade scaled up by the given amount."""
    new_grade = grade + amount
    if new_grade > 100:
        new_grade = 100
    return new_grade
```



an if statement (with no else case) is a 1-way conditional

- depending on the test condition, either execute the indented code or don't

an if-else statement (with else case) is a 2-way conditional

- depending on the test condition, execute one block of indented code or the other

18

If examples

one more revision: wind chill is not intended for temperatures ≥ 50°

- could add a check for temp ≥ 50, then return what? temp?

```
def wind_chill(temp, wind):
    """Calculates the wind chill factor."""
    if temp >= 50 or wind <= 3:
        return temp
    else:
        return 35.74 + 0.6215*temp + \
               (0.4275*temp - 35.75)*(wind**0.16)
```

really want to signify that the value is undefined

- the `float` function will convert a string into its corresponding number
  e.g., `float("12.5")` → `12.5`
- the expression `float("nan")` returns a special value, `nan`, that stands for 'not a number'
- whenever `nan` appears in an expression, the result is still `nan`

```
>>> x = float("12.5")
>>> x
12.5
>>> y = float("nan")
>>> y
nan
>>> z = y + 1
>>> z
nan
```

19

Cascading if-else

now have 3 different cases, so need a 3-way conditional

- can accomplish this by nesting if-else statements
- known as a *cascading if-else* (control cascades down from one test to the next)

```
def wind_chill(temp, wind):
    """Calculates the wind chill factor."""
    if temp >= 50:
        return float("nan")
    else:
        if wind <= 3:
            return temp
        else:
            return 35.74 + 0.6215*temp + \
                   (0.4275*temp - 35.75)*(wind**0.16)
```

reminder: Python uses indentation to determine code structure

- must make sure to align statements inside the appropriate if-else case

20

Cascading if-else: elif

because multi-way conditionals are fairly common, a variant exists to simplify the structure

- `elif` is shorthand for else-if
- introduces the next case without having to nest

```
def wind_chill(temp, wind):
    """Calculates the wind chill factor."""
    if temp >= 50:
        return float("nan")
    else:
        if wind <= 3:
            return temp
        else:
            return 35.74 + 0.6215*temp + \
                   (0.4275*temp - 35.75)*(wind**0.16)
```

```
def wind_chill(temp, wind):
    """Calculates the wind chill factor."""
    if temp >= 50:
        return float("nan")
    elif wind <= 3:
        return temp
    else:
        return 35.74 + 0.6215*temp + \
               (0.4275*temp - 35.75)*(wind**0.16)
```

21

Exercise: letter grades

define a Python function named `letter_grade`, that takes one input (a course average) and returns the corresponding letter grade

- assume grades of "A", "B", "C", "D", and "F" (no + or -)
- assume standard grade cutoffs

  e.g., `letter_grade(90)` should return "A"
  `letter_grade(89)` should return "B"

```
def letter_grade(average):
    ????
```

22