(U) TL;DR

(U) The key distinction between procedural and functional programming is this: **a procedural function may have side effects**--it may change the state of its inputs or something outside itself, giving you a different result when running it a second time. **Functional programming avoids side effects**, ensuring that functions don't modify anything outside themselves.

(U) Note

P.L. 86-36

(U) The contents of this notebook have been borrowed from the beginning of [redacted] essay, "A practical introduction to functional programming." A full notebook version of that essay can be found here. (Note that it uses Python 2.)

(U) Functional vs. Not

(U) The best way to understand side effects is with an example.

(U) This function is *not* functional:

```
a = 0
def increment():
    global a
    a += 1
```

(U) This function *is* functional:

```
def increment(a):
    return a + 1
```

(U) Map-Reduce

(U) Let's jump into functional coding. One common use is **map-reduce**, which you may have heard of. Let's see if we can make sense of it.

(U) map

(U) Conceptually, **map** is a function that takes two arguments: another function and a collection of items. It will

1. run the function on each item of the original collection and
2. return a new collection containing the results,

3. leaving the original collection unchanged.

(U) In Python 3, the input collection must simply be *iterable* (e.g. `list`, `tuple`, `string`). Its `map` function returns an *iterator* that runs the input function on each item of iterable.

(U) Example 1: Name Lengths

(U) Take a list of names and get a list of the name lengths:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(list(name_lengths))
```

(U) Example 2: Squaring

(U) Square every number in a list:

```
squares = map(lambda x: x * x, [0, 1, 2, 3, 4])
print(list(squares))
```

(U) A digression on `lambda`

(U) So what's going on with that input function? `lambda` will let you define and use an *unnamed* function. Arguments fit between the `lambda` and the colon while the stuff after the colon gets *implicitly* returned (i.e. without explicitly using a `return` statement).

(U) Lambdas are most useful when:

- your function is simple and
- you only need to use it once.

(U) Consider the usual way of defining a function:

```
def square(x):
    return x * x
```

```
square(4)
```

```
# we could have done this instead
squares = map(square, [0, 1, 2, 3, 4])
print(list(squares))
```

(U) Now let's define the same function using a lambda:

```
lambda x: x * x
```

(U) Fine, but how do we call that resulting function? Unfortunately, it's too late now; we didn't store the result, so it's lost in the ether.

(U) Let's try again:

```
ima_function_variable = lambda x: x * x
```

```
type(ima_function_variable)
```

```
ima_function_variable(4)
```

```
# be careful!
ima_function_variable = 'something else'
```

```
# our lambda function is gone again
ima_function_variable(4)
```

(U) Example 3: Code Names

(U) OK, back to `map`. Here's a procedural way to take a list of real names and replace them with randomly assigned code names.

```
import random
```

```
names = ['Mary', 'Isla', 'Sam']
code_names = ['Mr. Pink', 'Mr. Orange', 'Mr. Blonde']

for i in range(len(names)):
    names[i] = random.choice(code_names)

print(names)
```

(U) Here's the functional version:

```
names = ['Mary', 'Isla', 'Sam']
covernames = map(lambda x: random.choice(['Mr. Pink', 'Mr. Orange', 'Mr. Blonde']), names)
print(list(covernames))
```

(U) Exercise: Code Names...Improved?

(U) The procedural code below generates code names using a new method. Rewrite it using `map`.

```
names = ['Mary', 'Isla', 'Sam']

for i in range(len(names)):
    names[i] = hash(names[i])

print(names)
```

```
# your code here
```

reduce

(U) **Reduce** is the follow-on counterpart to map. Given a function and a collection of items, it uses the function to combine them into a single value and returns that result.

(U) The function passed to `reduce` has some restrictions, though. It must take two arguments: an *accumulator* and an *update* value. The update value is like it was before with `map`; it will get set to each item in the collection one by one. The accumulator is new. It will receive the output from the previous function call, thus "accumulating" the combined value from item to item through the collection.

(U) **Note**: in Python 2, `reduce` was a built-in function. Python 3 moved it into the `functools` package.

(U) Example

(U) Get the sum of all items in a collection.

```
import functools
```

```
sum = functools.reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(sum)
```

UNCLASSIFIED

Recursion Examples

(b)(3)-P.L. 86-36

0

Updated over 3 years ago by [redacted]

3 193 46

python examples recursion fcs6 fibonacci games ai

(U) Some simple recursion examples in Python

Recommendations

Recursion

Recursion provides a way to loop without loops. By calling itself on updated data, a recursive function can progress through a problem and traverse the options.

Nth Fibonacci Number

https://wikipedia.nsa.ic.gov/en/Fibonacci_number

This returns the nth Fibonacci number in the Fibonacci Sequence using recursion.

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

```
nth_fibonacci(10)
```

Fibonacci Sequence

This returns a list of the first n Fibonacci Numbers using recursion.

```
def fibonacci(n, seq=[]):
    if len(seq) == n:
        return seq
    elif len(seq) == 0:
        return fibonacci(n, [1])
    elif len(seq) == 1:
        return fibonacci(n, [1,1])
    else:
        next_value = seq[-2] + seq[-1]
        return fibonacci(n, seq + [next_value])
```

```
fibonacci(5)
```

Simple Game

This simple game just takes in a list of nine elements and tries to modify each slot until all the numbers from 1 to 9 are in the list.

```
import random
```

```
def improve(input_list, missing):
    random_index = random.choice(list(range(len(input_list))))
    random_value = random.choice(missing)
    new_list = input_list[:]
    new_list[random_index] = random_value
    return new_list

def find_missing(input_list):
    missing = [ x for x in list(range(1,10)) if x not in input_list ]
    return missing

def one_to_nine(input_list):
    print(input_list)
    missing = find_missing(input_list)

    if len(missing) == 0:
        return input_list
    else:
        new_list = improve(input_list, missing)
        return one_to_nine(new_list)
```

```
one_to_nine([1,1,1,1,1,1,1,1,1])
```

Simple Game Revised

This revision of the same simple game comes up with a list of possible improvements and tries to pick the best one to pursue. You'll notice that it takes fewer attempts to reach an anser than the original version of this simple game.

Doc ID: 6689695

```python
def improve(input_list, missing):
    random_index = random.choice(list(range(len(input_list))))
    random_value = random.choice(missing)
    new_list = input_list[:]
    new_list[random_index] = random_value
    return new_list

def find_missing(input_list):
    missing = [ x for x in list(range(1,10)) if x not in input_list ]
    return missing

def score(input_list):
    missing = find_missing(input_list)
    return (len(missing), input_list)

def best_scoring_list(scored_lists):
    lowest = 100
    best_list = []

    for x in scored_lists:
        score = x[0]
        input_list = x[1]

        if score < lowest:
            lowest = score
            best_list = input_list

    return best_list

def one_to_nine(input_list):
    print(input_list)
    missing = find_missing(input_list)

    if len(missing) == 0:
        return input_list
    else:
        possible_improvements = [ improve(input_list, missing) for i in range(len(missing)) ]
        scored_improvements = [ score(i) for i in possible_improvements ]
        best_list = best_scoring_list(scored_improvements)
        return one_to_nine(best_list)
```

```
one_to_nine([1,1,1,1,1,1,1,1,1])
```

Simple Game as a Tree

We can think of our strategy as a tree of options that we traverse, following branches that show that they're going to improve our chances of finding a solution. This is an extremely simplified form of what many video games use for their AI. We put our possible_improvements in a generator so they will only be created as needed. If we put them in a list comprehension as before, then all possible improvements would be generated even though many of them will likely go unused. In the end, we return any() with a generator for the branches. Since any() only needs one item to be True, it will return True as soon as a solution is found; when len(missing) == 0.

You'll notice that this results in more iterations of one_to_nine() than in the previous revision. However, the previous revision also generated a lot of data that ends up getting discarded. In other words, there's probably more processing and memory consumed by the previous revision behind the scenes.

```
def improve(input_list, missing):
    random_index = random.choice(list(range(len(input_list))))
    random_value = random.choice(missing)
    new_list = input_list[:]
    new_list[random_index] = random_value
    return new_list

def find_missing(input_list):
    missing = [ x for x in list(range(1,10)) if x not in input_list ]
    return missing

def one_to_nine(input_list, prev_missing=None):
    print(input_list)
    missing = find_missing(input_list)

    if prev_missing is None:
        return one_to_nine(input_list, missing)
    elif len(missing) == 0:
        return True
    elif len(missing) > len(prev_missing):
        return False
    else:
        possible_improvements = ( improve(input_list, missing) for i in range(len(missing)) )
        return any(( one_to_nine(p, missing) for p in possible_improvements ))
```

Doc ID: 6689695

```
one_to_nine([1,1,1,1,1,1,1,1,1])
```

Doc ID: 6689695

Module: Command Line Arguments

0

(b)(3)-P.L. 86-36

Updated almost 2 years ago by [redacted] in COMP 3321

3 323 123

python

(U) Module: Command Line Arguments

Recommendations

UNCLASSIFIED

(U) Most command line programs accept options and arguments, and many even provide help messages that indicate what options are available, and how they are to be used. For example, the utility program `mv` takes two arguments, and most often moves the first argument (the source) to the second (the destination). It has other ways of operating, which are enabled by optional flags and arguments; from a command prompt, type `mv --help` to see more.

(U) There are several ways to enable this type of functionality in a Python program, and the best way to do it has been a source of contention. In particular, this lesson will cover the **argparse**module, which was added to the standard library in Python 2.7, and not the **optparse** module which was deprecated at that time.

(U) Everything passed as arguments to a Python program is available in the interpreter as the list of strings in `sys.argv` . In an interactive session, `sys.argv` always starts out as `['']` . When running a script, `sys.argv[0]` is the name of the script. We start by examining what `sys.argv` looks like. Put the following commands in a file called `argtest.py` or similar:

```
import sys
print(sys.argv)
```

```
# ...or make python do it!
contents = '''import sys
print(sys.argv)
'''

with open('argtest.py', 'w') as f:
    f.write(contents)
```

(U) Close the file and execute it from the command line with some arguments:

```
# the '!' at the beginning tells jupyter to send what follows to the command line
!python3 argtest.py -xzf --v foo --othervar=bar file1 file2
```

```
# => ['argtest.py', '-xzf', '--v', 'foo', '--othervar=bar', 'file1', 'file2']
```

(U) In all of the argument parsing that follows, `sys.argv` will be involved, although that may happen either implicitly or explicitly. Although it is often unwise to do so within a script, `sys.argv` can be modified, for instance during testing within an interactive session.

(U) Note that in Jupyter you still have `argv`, but it may not be what you expect. If you look at it, you'll see how this Python 3 kernel is being called:

```
import sys
print(sys.argv)
```

(U) The Hard Way: **getopt**

(U) For programs with only simple arguments, the **getopt** module provides functionality similar to the `getopt` function in C. The main method in the module is `getopt`, which takes a list of strings, usually `sys.argv[1:]` and parses it according to a string of options, with optional *long options*, which are allowed to have more than one letter; explanations are best left to examples. This method returns a pair of lists, one containing `(option, value)` tuples, the other containing additional positional arguments. These values must then be further processed within the program; it might be useful, for instance, to put the `(option, value)` tuples into a `dict`. If `getopt` receives an unexpected option, it throws an error. If it does not receive all the arguments it requests, no error is thrown, and the missing arguments are not present in the returned value.

```
import getopt
```

```
getopt.getopt('-a arg'.split(),'a:') # a expects an argument
```

```
getopt.getopt('-a arg'.split(), 'a:b') # no b, no problem
```

```
getopt.getopt('-b arg -a my-file.txt'.split(), 'ab:') # my-file.txt is argument, not option
```

```
getopt.getopt('-a arg --output=other-file.txt my-file.txt'.split(),'a:b',['output=']) # long options
```

(U) For programs that use `getopt`, usage help must be provided manually.

```
def usage():
    print("""usage: my_program.py -[abh] file1, file2, ...""")
```

```
# this won't actually find anything in Jupyter, since ipython3 probably doesn't have these options
opts, args = getopt.getopt(sys.argv[1:], 'abh')
```

```
opt_dict = dict(opts)
```

```
if '-h' in opt_dict:
    usage()
```

(U) The **argparse** Module

(U) Integrated help is one of the benefits of **argparse**, along with the ability to specify both short and long options versions of arguments. There is some additional complication in setting up **argparse**, but it is the right thing to do for all but the most simple programs.

(U) Basic Usage

(U) The main class in **argparse** is `ArgumentParser`. After an `ArgumentParser` is instantiated, arguments are added to it with the `add_argument` method. After all the arguments are added, the `parse_args` method is called. By default, it reads from `sys.argv[1:]`, but can also be passed a list of strings, primarily for testing. Both positional arguments (required) and optional arguments indicated by flags are supported. An example will illustrate the operation.

```
import argparse
```

```
parser = argparse.ArgumentParser()
```

```
parser.add_argument('n')
```

```
parser.add_argument('-f')
```

```
parser.add_argument('-i','--input')
```

```
parser.print_help()
```

```
parser.parse_args('abc -f xyz'.split())
```

```
parser.parse_args('-f xyz abc --input=myfile.txt'.split())
```

```
parser.parse_known_args('-f xyz abc --input=myfile.txt -o otherfile.txt'.split())
```

```
args = parser.parse_args('-f xyz abc --input=myfile.txt'.split())
```

```
args.f
```

(U) As seen in the final two lines, positional arguments and optioned arguments can come in any order, which is not the case with **getopt**. If multiple positional arguments are specified, they are parsed in the order in which they were added to the `ArgumentParser`. The object returned by `parse_args` is called a `Namespace`, but it is just an object which contains all the parsed data. Unless otherwise specified, the attribute names are derived from the option names. Positional arguments are used directly, while short and long flags have leading hypens stripped and internal hyphens converted to underscores. If more than one flag is specified for an argument, the first long flag is used if present; otherwise, the first short flag is used.

(U) Here is how argparse could look in your code.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('n')
parser.add_argument('-f')
parser.add_argument('-i','--input')
args = parser.parse_args()

print(args)```
```

```
contents = '''import argparse
parser = argparse.ArgumentParser()
parser.add_argument('n')
parser.add_argument('-f')
parser.add_argument('-i','--input')
args = parser.parse_args()

print(args)
'''

with open('argparsetest.py', 'w') as f:
    f.write(contents)
```

(U) Now we can simulate running it:

```
!python3 argparsetest.py -f xyz abc --input=myfile.txt
```

```
!python3 argparsetest.py -h
```

(U) Advanced Options

(U) The `add_argument` method supports a large number of keyword arguments that configure behavior more finely than the defaults. For instance, the `type` argument will make the parser attempt to convert arguments to the specified type, such as `int`, `float`, or `file`. In fact, you could use any class for the type, as long as it can be constructed from a single string argument.

```
parser = argparse.ArgumentParser()
```

```
parser.add_argument('n', type=int)
```

```
parser.parse_args('5'.split())
```

(U) The `nargs` keyword lets an argument specify a fixed or variable number of arguments to consume, which are then stored into a list. This applies to both positional and optional arugments. Giving `nargs` the value `'?'` makes positional arguments optional, in which case the `default` keyword is required.

```
parser = argparse.ArgumentParser()
```

```
parser.add_argument('n', nargs=2)
```

```
parser.add_argument('-m', nargs='*') # arbitrary arguments
```

```
parser.parse_args('n1 n2 -m a b c'.split())
```

```
parser.add_argument('o', nargs='?', default='OoO')
```

```
parser.parse_args('n1 n2 oOo -m a b c'.split())
```

(U) The `default` keyword can also be used with optional arguments. When an optional argument is always used as as a flag without parameters, it is also possible to use the `action='store_const'` and `const` keywords. In this case, when the option is detected, the `Namespace` is given an appropriately-named attribute with `const` as its value. If the option is not present in the parsed args, the attribute is created with the value given in `default`, or `None` if `default` isn't set.

```
parser = argparse.ArgumentParser()
```

```
parser.add_argument('-n', action='store_const', const=7)
```

```
parser.add_argument('-b', action='store_true')
```

```
parser.add_argument('-c', action='store_const', const=5, default=3)
```

```
parser.parse_args([])
```

```
parser.parse_args('-n -b'.split())
```

```
parser.parse_args('-cbn'.split())
```

(U) The `action` keyword can take other arguments; for instance, `action='store_true'` and `action='store_false'` can be used instead of setting `const` to a boolean value.

(U) Once again, we have only scratched the surface of a module, **argparse** in this case. Check out the documentation for more details (e.g. changing attribute names with the `dest` keyword, writing custom action functions, providing different parsers for subprograms).

Module: Dates and Times

(b)(3)-P.L. 86-36

0

Updated 10 months ago by [redacted] in COMP 3321

3 2 450 204

fcs6 python datetime arrow parsedate

(U) How to manipulate dates and times in Python.

Recommendations

UNCLASSIFIED

(U) Introduction

(U) There are many great built-in tools for date and time manipulation in Python. They are spread over a few different modules, which is a little annoying.

(U) That being said, the **datetime** module is very complete and the most useful, so we will concentrate on that one that most. The other one that has some nice methods is the **time** module.

(U) `time` Module

(U) The **time** module is handy for its time accessor functions and `sleep` command. It is most useful when you want quick *access* to the time but don't need to *manipulate* it.

```
import time
```

```
time.time()  # "epoch" time (seconds since Jan. 1st, 1970)
```

```
time.gmtime()
```

```
time.localtime()
```

```
time.gmtime() == time.localtime()
```

```
time.asctime()  # will take an optional timestamp
```

```
time.strftime('%c')  # many formatting options here
```

```
time.strptime('Tue Nov 19 07:04:38 2013')
```

(U) The last method you might use from the time module is `sleep`. (Doesn't this seem out of place?)

```
time.sleep(10)   # argument is a number of seconds
print("I'm awake!")
```

(U) `datetime` Module

(U) The **datetime** module is a more robust module for dates and times that is object-oriented and has a notion of datetime arithmetic.

(U) There are 5 basic types that comes with the datetime module. These are:

1. `date` : a type to store the date (year, month, day) using the current Gregorian calendar,
2. `time` : a type to store the time (hour, minute, second, microsecond, tzinfo--all idealized, with no notion of leap seconds),
3. `datetime` : a type to store both date and time together,
4. `timedelta` : a type to store the duration or difference between two date, time, or datetime instances, and
5. `tzinfo` : a base class for storing and using time zone information (we will not look at this).

(U) `date` type

```
import datetime
```

```
datetime.date(2013, 11, 19)
```

```
datetime.date.today()
```

```
datetime.date.fromtimestamp(time.time())
```

```
today = datetime.date.today()
```

```
today.day
```

```
today.month
today.timetuple()
today.weekday()   # 0 ordered starting from Monday
today.isoweekday() # 1 ordered starting from Monday
print(today)
today.strftime('%d %m %Y')
```

(U) time type

```
t = datetime.time(8, 30, 50, 0)
t.hour = 9
t.hour
t.minute
t.second
print(t)
print(t.replace(hour=12))
t.hour = 8
print(t)
```

(U) datetime type

```
dt = datetime.datetime(2013, 11, 19, 8, 30, 50, 0)
print(dt)
datetime.datetime.fromtimestamp(time.time())
```

```
now = datetime.datetime.now()
```

(U) We can break apart the `datetime` object into `date` and `time` objects. We can also combine `date` and `time` objects into one `datetime` object.

```
now.date()
print(now.date())
print(now.time())
day = datetime.date(2011, 12, 30)
t = datetime.time(2, 30, 38)
day
t
dt = datetime.datetime.combine(day,t)
print(dt)
```

(U) `timedelta` type

(U) The best part of the **datetime** module is date arithmetic. What do you get when you subtract two dates?

```
day1 = datetime.datetime(2013, 10, 30)
day2 = datetime.datetime(2013, 9, 20)
day1 - day2
print(day1 - day2)
print(day2 - day1)
print(day1 + day2)  # Of course not...that doesn't make sense :)
```

(U) The `timedelta` type is a measure of duration between two time events. So, if we subtract two datetime objects (or `date` or `time`) as we did above, we get a `timedelta` object. The properties of a `timedelta` object are `(days, seconds, microseconds, milliseconds, minutes, hours, weeks)` . They are all optional and set to 0 by default. A `timedelta` object can take these values as arguments but converts and normalizes the data into days, seconds, and microseconds.

```
from datetime import timedelta
```

```
day = timedelta(days=1)
```

```
day
```

```
now = datetime.datetime.now()
```

```
now + day
```

```
now - day
```

```
now + 300*day
```

```
now - 175*day
```

```
year = timedelta(days=365)
```

```
another_year = timedelta(weeks=40, days=84, hours=23, minutes=50, seconds=600)
```

```
year == another_year
```

```
year.total_seconds()
```

```
ten_years = 10*year
ten_years
```

(U) Conversions

(U) It's easy to get confused when attempting to convert back and forth between strings, numbers, `time` s, and `datetime` s. When you need to do it, the best course of action is probably to open up an interactive session, fiddle around until you have what you need, then capture that in a well-named function. Still, some pointers may be helpful.

(U) Objects of types **time** and **datetime** provide `strptime` and `strftime` methods for converting times and dates from strings (a.k.a. ***p***arsing) and converting to strings (a.k.a. ***f***ormatting), respectively. These methods employ a custom syntax that is shared across many programming languages.

```
print(ga_dt)
```

(U) Localize the time using the Georgia timezone, then convert to Kabul timezone

```
kabul_tz = pytz.timezone('Asia/Kabul')
kabul_dt = ga_tz.localize(ga_dt).astimezone(kabul_tz)
print(kabul_tz.normalize(kabul_dt))
```

(U) To get a list of all timezones:

```
pytz.all_timezones
```

(U) The arrow package

(U) The arrow package is a third party package useful for manipulating dates and times in a non-naive way (in this case, meaning that it deals with multiple timezones very seamlessly). Examples below are based on examples from "20 Python Libraries You Aren't Using (But Should)" on Safari.

```
import ipydeps
ipydeps.pip('arrow')
import arrow
```

```
t0 = arrow.now()
print(t0)

t1 = arrow.utcnow()
print(t1)

difference = (t0 - t1).total_seconds()

print('Total difference: %.2f seconds' % difference)
```

```
t0 = arrow.now()
t0
```

```
t0.date()
```

```
t0.time()
```

```
t0.timestamp
```

```
t0.year
```

```
t0.month
```

```
t0.day
```

```
t0.datetime
```

```
t1.datetime
```

```
t0 = arrow.now()
t0.humanize()
```

```
t0.humanize()
```

```
t0 = t0.replace(hours=-3,minutes=10)
t0.humanize()
```

(U) The parsedate module

(U) The parsedate package is a third party package that does a very good job of parsing dates and times from "messy" input formats. It also does a good job of calculating relative times from human-sytle inpu. Examples below are from "20 Python Libraries You Aren't Using (But Should)" on Safari.

```
ipydeps.pip('parsedatetime')
import parsedatetime as pdt
```

Doc ID: 6689695

```
cal = pdt.Calendar()

examples = [
    "2016-07-16",
    "2016/07/16",
    "2016-7-16",
    "2016/7/16",
    "07-16-2016",
    "7-16-2016",
    "7-16-16",
    "7/16/16",
]

#print the header
print('{:30s}{:>30s}'.format('Input', 'Result'))
print('=' * 60)

#loop through the examples list and show that parseDT successfully parses out the date/time based on the messy values
for e in examples:
    dt, result = cal.parseDT(e)
    print('{:<30s}{:>30}'.format('"' + e + '"', dt.ctime()))
```

Doc ID: 6689695

```
examples = [
    "19 November 1975",
    "19 November 75",
    "19 Nov 75",
    "tomorrow",
    "yesterday",
    "10 minutes from now",
    "the first of January, 2001",
    "3 days ago",
    "in four days' time",
    "two weeks from now",
    "three months ago",
    "2 weeks and 3 days in the future",
]

#print the time right now for reference
print('Now: {}'.format(datetime.datetime.now().ctime()), end='\n\n')

#print the header
print('{:40s}{:>30s}'.format('Input', 'Result'))
print('=' * 70)

#loop through the examples list to show how parseDT can successfully determine the date/time based on both messy input
# and messy relative time offset inputs
for e in examples:
    dt, result = cal.parseDT(e)
    print('{:<40s}{:>30}'.format('"' + e + '"', dt.ctime()))
```

UNCLASSIFIED

Doc ID: 6689695

COMP3321 Datetime Exercises

(b)(3)-P.L. 86-36

0

Created almost 3 years ago by [redacted] in COMP 3321

3 143 23

python exercises

(U) Datetime Exercises for COMP3321

Recommendations

(U) Datetime Exercise

(U) How long before Christmas?

(U) How many seconds since you were born?

(U) What is the average number of days between Easter and Christmas for the years 2000 - 2999?

(U) What day of the week does Christmas fall on this year?

(U) You get a intercepted email with a POSIX timestamp of 1435074325. The email is from the leader of a Zendian extremist group and says that there will be an attack on the Zendian capitol in 14 hours. In Zendian local time, when will the attack occur? (Assume Zendia is in the same time zone as Kabul)

Doc ID: 6689695

Module: Interactive User Input with ipywidgets

0

(b)(3)-P.L. 86-36

Updated almost 2 years ago by [redacted] in COMP 3321

3 393 200

fcx93 ipywidgets python tutorial input

(U) Covers the ipywidgets library for getting interactive user input in Jupyter

Recommendations

UNCLASSIFIED//~~FOR OFFICIAL USE ONLY~~

(U) ipywidgets

(U) ipywidgets is used for making interactive widgets inside your jupyter notebook

(U) The most basic way to get user input is to use the python built in `input` function. For more complicated types of interaction, you can use ipywidgets

```
#input example (not using ipywidgets)
a=input("Give me your input: ")
print("your input was: "+a)
```

```
import ipywidgets
from ipywidgets import *
```

`interact` is the easiest way to get started with ipywidgets by creating a user interface and automatically calling the specified function

```
def f(x):
    return x*2

interact(f,x=10)
```

```
def g(check,y):
    print("{} {}".format(check,y))
interact(g,check=True,y="Hi there!")
```

But, if you need more flexibility, you can start from scratch by picking a widget and then calling the functionality you want. Hint: you get more widget choices this way.

```
IntSlider()
```

```
w=IntSlider()
```

```
w
```

You can explicitly display using IPython's display module. Note what happens when you display the same widget more than once!

```
from IPython.display import display
display(w)
```

```
w.value
```

Now we have a value from our slider we can use in code. But what other attributes or "keys" does our slider widget have?

```
w.max
```

```
new_w=IntSlider(max=200)
display(new_w)
```

You can also close your widget

```
w.close()
new_w.close()
```

Here are all the available widgets:

```
Widget.widget_types
```

Categories of widgets

Numeric: IntSlider, FloatSlider, IntRangeSlider, FloatRangeSlider, IntProgress, FloatProgress, BoundedIntText, BoundedFloatText, IntText, FloatText

Boolean: ToggleButton, Checkbox, Valid

Doc ID: 6689695

Selection: Dropdown, RadioButtons, Select, ToggleButtons, SelectMultiple

String Widgets: Text, Textarea

Other common: Button, ColorPicker, HTML, Image

```
Dropdown(options=["1","2","3","cat"])
```

```
bt = Button(description="Click me!")
display(bt)
```

Buttons don't do much on their own, so we have to use some event handling. We can define a function with the desired behavior and call it with the buttons `on_click` method.

```
def clicker(b):
    print("Hellow World!!!!")

bt.on_click(clicker)
```

```
def f(change):
    print(change['new'])

w = IntSlider()
display(w)
w.observe(f,names='value')
```

Wrapping Multiple Widgets in Boxes

When working with multiple input widgets, it's often nice to wrap it all in a nice little box. `ipywidgets` provides a few options for this--we'll cover `HBox` (horizontal box) and `VBox` (vertical box).

HBox

This will display the widgets horizontally

```
fruit_list = Dropdown(
  options = ['apple', 'cherry', 'orange', 'plum', 'pear']
)
fruit_label = HTML(
  value = 'Select a fruit from the list:  '
)
fruit_box = HBox(children=[fruit_label, fruit_list])
fruit_box
```

VBox

This will display the widgets (or boxes) vertically

```
num_label = HTML(
  value = 'Choose the number of fruits:  '
)

num_options = IntSlider(
  min=1,
  max=20
)

num_box = HBox(children=(num_label, num_options))

type_label = HTML(
  value = 'Select the type of fruit:  '
)

type_options = RadioButtons(
  options=('Under-ripe', 'Ripe', 'Rotten')
)

type_box = HBox(children=(type_label, type_options))


fruit_vbox = VBox(children=(fruit_box, num_box, type_box))
fruit_vbox
```

Specify Layout of the Widgets/Boxes

```
form_item_layout = Layout(
  display='flex',
  flex_flow='row',
  justify_content='space-between',
  width='70%',
  align_items='initial',
)

veggie_label = HTML(
  value = 'Select a vegetable from the list:  ',
  layout=Layout(width='20%', height='65px')
)

veggie_options = Dropdown(
  options=['corn', 'lettuce', 'tomato', 'potato', 'spinach'],
  layout=Layout(width='30%', height='65px')
)

veggie_box = HBox(children=(veggie_label, veggie_options),
                  layout=Layout(width='100%', border='solid 1px',
                                height='100px'))
veggie_box
```

Retrieving Values from a Box

```
box_values = {}
# the elements in a box can be accessed using the children attribute
for index, box in enumerate(fruit_vbox.children):
  for child in box.children:
    if type(child) != ipywidgets.widgets.widget_string.HTML:
      if index == 0:
        print("The selected fruit is: ", child.value)
        box_values['fruit'] = child.value
      elif index == 1:
        print("The select number of fruits is: ", str(child.value))
        box_values['count'] = child.value
      elif index == 2:
        print("The selected type of fruit is: ", str(child.value))
        box_values['type'] = child.value
box_values
```

Doc ID: 6689695

Module: GUI Basics with Tkinter

(b)(3)-P.L. 86-36

0

Updated almost 2 years ago by [redacted] in COMP 3321

3 5 592 225

python fcs6 comp3321

(U) Module: GUI Basics with Tkinter

Recommendations

UNCLASSIFIED~~//FOR OFFICIAL USE ONLY~~

(U) Tkinter

(U) Tkinter comes as part of Python, so is readily available for use--just import it. **Note**: While Tkinter is almost always available, Python can be installed without it, e.g. if Tcl/Tk is not available when Python is compiled. Tk is a widget library that was originally designed for the Tcl scripting language, but now has been ported to Perl, Ruby, Python, C++ and more.

(U) **NOTE**: In Python 2 it must be used as `Tkinter` with a capital `T`.

P.L. 86-36

(U) Setup

(U~~//FOUO)~~ This lesson cannot currently be run from Jupyter on LABBENCH due to displayback limitations. It should work using Anaconda's Jupyter locally [redacted] The examples can also be copied to files and run as scripts from MACHINESHOP if your display is properly configured.

(U~~//FOUO)~~ On MACHINESHOP

(U~~//FOUO)~~ To display Python Tk objects back from MACHINESHOP, you will need to run the following on your MASH instance. [redacted]

[redacted] **NOTE**: Not all steps may be necessary; verification needed.

```
yum -y groupinstall desktop
yum -y install tigervnc-server
yum -y install xrdp
/sbin/service xrdp start
chkconfig xrdp on
/sbin/service iptables stop
```

(U) What's a GUI (Graphical User Interface)?

(U) We all use them, some of us love them and hate them. Do we consider it 2- or 3-dimensional (2.5-dimensional)? Let's look at some very basic examples.

(U) Example 1

```
import tkinter as tk
root = tk.Tk()
root.mainloop()
```

We just created our first gui! But it doesn't do a whole lot yet. That is because we only created a blank/empty window that is waiting for our creation. `tk.Tk()` is the top level window that we will create for every gui that we make.

Parts of a gui:
Choose widgets ==> Arrange in window ==> Add functionality

(U) Example 2

A first look at widgets!

```
#Basic gui with a label and a button

import tkinter as tk
root = tk.Tk()

label = tk.Label(root, text="I am a label widget") #Create label
button = tk.Button(root, text="I am a button")     #create button

label.pack()                                       #Add label to gui
button.pack()                                      #Add button to gui
root.mainloop()
```

(U) Widget Types

Type	Description
`Button`	Users click on buttons to trigger some action. Button clicks can be translated into actions taken by your program. `Button` s usually display text but can show graphics.
`Canvas`	A surface on which you can draw graphs and/or plots and also use as the basis of your own widgets.
`CheckButton`	A special type of `Button` that has two states; clicking changes the state of the button from one to the other.
`Entry`	Used to enter single lines of text and all kinds of input.
Frame	A container for other widgets. One can set the border and background color and place other widgets in it.
`Label`	Used to display pieces of text or images, usually ones that won't change during the execution of the application.

Type	Description
`Listbox`	Used to display a set of choices. The user can select a single item or multiple items from the list. The `Listbox` can be also rendered as a set of radio buttons or checkboxes.
`Message`	Similar to `Text` but can automatically wrap text to a particular width and height.
`Menu`	Used to put a menu in your window if you need it. It corresponds to the menu bar at the top but can also be used as a pop-up.
`Menubutton`	Adds choices to your `Menu` s.
`Radiobutton`	Represents one of a set of mutually exclusive choices. Selecting one `Radiobutton` from a set deselects any others.
`Scale`	Lets the user set numeric values by dragging a slider.
`Scrollbar`	Implements scrolling on a larger widget such as a `Canvas` , `Listbox` , or `Text` .
`Text`	A multi-line formatted text widget that allows the textual content to be "rich." It may also contain embedded images and `Frame` s.
Toplevel	A special kind of `Frame` that interacts directly with the window manager. `Toplevel` s will usually have a title bar and features to interact with the window manager. The windows you see on your screen are mostly `Toplevel` windows, and your application can create additional `Toplevel` windows if it is set to do that.

Other widgets: `OptionMenu` , `LabelFrame` , `PanedWindow` , `Bitmap Class` , `Spinbox` , `Image Class` .

(U) Example 3

Let's look at some other widget examples.
Also notice, that widgets have their own special "widget variables." Instead of using builtin python types, tk widgets use their own objects for storing this internal information. The tk widget variables are: `StringVar` , `IntVar` , `DoubleVar` , and `BooleanVar`

Doc ID: 6689695

```python
import tkinter as tk

root = tk.Tk()

tk.Label(root, text="Enter your Password:").pack()
tk.Button(root, text="Search").pack()

v = tk.IntVar()
tk.Checkbutton(root, text="Remember Me", variable=v).pack()
tk.Entry(root, width=30)

v2 = tk.IntVar()
tk.Radiobutton(root, text="Male", variable=v2, value=1).pack()
tk.Radiobutton(root, text="Female", variable=v2, value=2).pack()

var = tk.IntVar()
tk.OptionMenu(root, var, "Select Country", "USA", "UK", "India", "Others").pack()
tk.Scrollbar(root, orient='vertical').pack()

root.mainloop()
```

(U) Three ways to configure a widget

1. (U) Setting the values during initialization. (The way we have been doing it so far).
2. (U) Using keys to set the values.
3. (U) Using the widget's `configure` method.

(U) Widget Attributes

(U) There are tons of options that can be set, but here are a few of the important ones.

Attribute	Description
`background / bg`	The color of the body of the widget (e.g. `'red'` , `'blue'` , `'green'` , `'black'`).
`foreground / fg`	The color used for text.
`padx, pady`	The amount of padding to put around the widget horizontally and vertically. Without these the widget will be just big enough for its content.
`borderwidth`	Creates a visible border around a widget.
`height, width`	Specifies the height and width of the widget.
`disableforeground`	When a widget is disabled, this is the color of its text (usually gray).
`State`	Default is `'normal'` but also can use `'disabled'` or `'active'`

(U) Example 4

Let's try redoing Example 2 using the key/value method to make our label and the `.configure()` method to make our button.

```
#Basic gui with a label and a button

import tkinter as tk
root = tk.Tk()

label = tk.Label(root)
label["text"]="I am a label widget"        #using keys
button = tk.Button(root)
button.configure(text="I am a button")     #using configure

label.pack()                                    #Add label to gui
button.pack()                                   #Add button to gui
root.mainloop()
```

(U) Geometry Managers

(U) Now that we know how to create widgets, we're on to step two: arranging them in our window!

(U) There are mainly two types of geometry managers:

- `Pack`
- `Grid`

(U) There is also a third--Place--but maybe that's not for today.

(U) Pack

(U) Quick, easy, effective. If things get complicated, use `Grid` instead.

Attribute	Description
`fill`	Can be `X`, `Y`, or `Both`. `X` does the horizontal, `Y` does the vertical.
`expand`	`False` means the widget is never resized, `True` means the widget is resized when the container is resized.
`side`	Which side the widget will be packed against (`TOP`, `BOTTOM`, `RIGHT` or `LEFT`).

(U) Example 5

The pack geometry manager arranges widgets relative to window/frame you are putting them in. For example, if you select `side=LEFT` it will pack you widget against the left side of the widget.

```
#Example using pack geomtry manager

from tkinter import *
root = Tk()

parent = Frame(root)
# placing widgets top-down
Button(parent, text='ALL IS WELL').pack(fill=X)
Button(parent, text='BACK TO BASICS').pack(fill=X)
Button(parent, text='CATCH ME IF U CAN').pack(fill=X)
# placing widgets side by side
Button(parent, text='LEFT').pack(side=LEFT)
Button(parent, text='CENTER').pack(side=LEFT)
Button(parent, text='RIGHT').pack(side=LEFT)
parent.pack()
root.mainloop()
```

(U) Example 6

Generally, the pack geometry manager is best for simple gui's, but one way to make more complicated gui's using `pack` is to group widgets together in a `Frame` and then add the `Frame` to your window.

```
#Example using pack geometry manager

from tkinter import *
root = Tk()

frame = Frame(root)      #Add frame for grouping widgits

# demo of side and fill options
Label(frame, text="Pack Demo of side and fill").pack()
Button(frame, text="A").pack(side=LEFT, fill=Y)
Button(frame, text="B").pack(side=TOP, fill=X)
Button(frame, text="C").pack(side=RIGHT, fill=NONE)
Button(frame, text="D").pack(side=TOP, fill=BOTH)
frame.pack()
# note the top frame does not expand nor does it fill in

# X or Y directions
# demo of expand options - best understood by expanding the root widget and seeing the effect on all the three buttons below.
Label (root, text="Pack Demo of expand").pack()
Button(root, text="I do not expand").pack()
Button(root, text="I do not fill x but I do not expand").pack(expand = 1)
Button(root, text="I fill x and expand").pack(fill=X, expand=1)
root.mainloop()
```

(U) Grid

(U) Use it when things get complicated, but it can be complicated in itself!

Attribute	Description
`row`	The row in which the widget should appear.
`column`	The column in which the widget should appear.
`sticky`	Can be `N`, `S`, `E`, or `W`. One needs to actually see this work, but it's important if you want the widget to resize with everything else.
`rowspan, columnspan`	Widgets can start in one widget and occupy more than one row or column.

(U) Rowconfigure and columnconfigure

(U) Most layout definitions start with these.

Option	Description
`minsize`	Defines the row's or column's minimum size.
`pad`	Sets the size of the row or column by adding the specified amount of padding to the height of the row or the width of the column.
`weight`	Determines how additional space is distrbuted between the rows and columns as the frame expands. The higher the weight, the more of the additional space is taken up. A row weight of 2 will expand twice as fast as that of 1.

(U) Example 7

The `grid` geometry manager starts with row zero and column zero up in the top left hand corner of your window. When you add a widget using `grid`, the default is `row=0` and `column=0` so you don't have to explicitly state it, although it is good practice.

```
#Basic example using grid geometry manager

from tkinter import *
root = Tk()
Label(root, text="Username").grid(row=0, sticky=W)
Label(root, text="Password").grid(row=1, sticky=W)
Entry(root).grid(row=0, column=1, sticky=E)
Entry(root).grid(row=1, column=1, sticky=E)
Button(root, text="Login").grid(row=2, column=1, sticky=E)
root.mainloop()
```

(U) Example 8

You could create this example using `pack` ... But it you would probably need a lot of frames. Using `grid` for something like this is much easier!

```
#More advanced example using geometry manager

from tkinter import *
parent = Tk()
parent.title('Find & Replace')  #Title of window

#First label and text entry widgits
Label(parent, text="Find:").grid(row=0, column=0, sticky='e')
Entry(parent, width=60).grid(row=0, column=1, padx=2, pady=2, sticky='we', columnspan=9)

#second label and text entry widgits
Label(parent, text="Replace:").grid(row=1, column=0, sticky='e')
Entry(parent).grid(row=1, column=1, padx=2, pady=2, sticky='we', columnspan=9)

#buttons
Button(parent, text="Find").grid(
row=0, column=10, sticky='e' + 'w', padx=2, pady=2)
Button(parent, text="Find All").grid(
  row=1, column=10, sticky='e' + 'w', padx=2)
Button(parent, text="Replace").grid(row=2, column=10, sticky='e' + 'w', padx=2)
Button(parent, text="Replace All").grid(
  row=3, column=10, sticky='e' + 'w', padx=2)

#Checkboxes
Checkbutton(parent, text='Match whole word only').grid(
  row=2, column=1, columnspan=4, sticky='w')
Checkbutton(parent, text='Match Case').grid(
  row=3, column=1, columnspan=4, sticky='w')
Checkbutton(parent, text='Wrap around').grid(
  row=4, column=1, columnspan=4, sticky='w')

#Label and radio buttons
Label(parent, text="Direction:").grid(row=2, column=6, sticky='w')
Radiobutton(parent, text='Up', value=1).grid(
  row=3, column=6, columnspan=6, sticky='w')
Radiobutton(parent, text='Down', value=2).grid(
  row=3, column=7, columnspan=2, sticky='e')

#run gui
parent.mainloop()
```

(U) Object Oriented GUI's

Up until now, all our examples show how to use tkinter with out creating classes, but in reality, GUI programs usually get very large very quickly. To keep things organized, it's best to encapsulate your GUI in a class and group your widget creation inside of class functions.

(U) Example 9

Wrapping a GUI inside a class. Note that in real life, you probably won't be running a GUI from inside Jupyter. This example shows how to check if you are in `main`, which you will need if you are running from the command line.

```
from tkinter import *


class Application(Frame):
    '''
    class Application :: Basic Tkinter example
    '''

def create_widgets(self):
        #Create Widget
        self.hi_there = Button(self)
        self.hi_there['text'] = 'hello'
        self.hi_there['fg'] = 'blue'

        #Arrange widget
        self.hi_there.pack({'side':'left'})


        #Create Widget
        self.QUIT = Button(self)
        self.QUIT['text'] = 'Quit'
        self.QUIT['fg'] = 'red'

        #Arrange widget
        self.QUIT.pack({'side':'left'})


    def __init__(self, master = None):
        '''
        Constructor
        '''
        Frame.__init__(self,master)
        self.pack()
        self.create_widgets()


def main():
    root = Tk()
    app = Application(master=root)
    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Callbacks and Eventing

(U)Most widgets have a `command` attribute to associate a callback function for when the widget is clicked.

When you need your gui to respond to something other than a mouse click or a specific kind of mouse click, you can bind you widget to an event.

(U) Example 9.1

Add a simple callback function to example 9 using.

What does the `print` command do when you click the "hello" button?

What happens when you click the "Quit" button?

Hint: you really are "quitting" your program. It just doesn't destroy (that is, close) your window!

Doc ID: 6689695

```python
from tkinter import *

class Application(Frame):
    '''
    class Application :: Basic Tkinter example
    '''

    #Callback function
    def say_hello(self):
        print('Hello There')

    def create_widgets(self):
        #Create Widget
        self.hi_there = Button(self)
        self.hi_there['text'] = 'hello'
        self.hi_there['fg'] = 'blue'

        #Arrange widget
        self.hi_there.pack({'side':'left'})
        #call back functionality
        self.hi_there['command'] = self.say_hello


        #Create Widget
        self.QUIT = Button(self)
        self.QUIT['text'] = 'Quit'
        self.QUIT['fg'] = 'red'

        #Arrange widget
        self.QUIT.pack({'side':'left'})
        #call back functionality
        self.QUIT['command'] = self.quit


    def __init__(self, master = None):
        '''
        Constructor
        '''
        Frame.__init__(self,master)
        self.pack()
        self.create_widgets()


def main():
    root = Tk() #create window
    app = Application(master=root)
    app.mainloop()
```

```
if __name__ == '__main__':
    main()
```

(U) Example 9.2

(U) Using the key/value method made it easy to check we were following the pattern:

1. Choose widget
2. Add to window/arrange
3. Add functionality

But it is shorter to write the code using the initialize method. Even if your code isn't written in the order of these steps, this is still the order you want to think about them.

```
from tkinter import *


class Application(Frame):
    '''
    class Application :: Basic Tkinter example
    '''
    def say_hello(self):
        print('Hello There')

    def create_widgets(self):
        self.hi_there = Button(self, text='hello',fg='blue',command=self.say_hello)
        self.hi_there.pack(side='left')

        self.QUIT = Button(self,text='quit',fg='red',command=self.quit)
        self.QUIT.pack(side='left')

    def __init__(self, master = None):
        '''
        Constructor
        '''
        Frame.__init__(self,master)
        self.pack()
        self.create_widgets()


def main():
    root = Tk()
    app = Application(master=root)
    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Example 10

(U) Now that we have covered the three basics of GUI's: selecting widgets, arranging them in our window, and adding functionality, let's try a more complicated example.

Doc ID: 6689695

```python
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #----------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()


    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #-------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
        #-------------------------------------------------------------

    def colorgen(self):
        '''
        Generator function that alternates between red and blue
        '''
        while True:
            yield 'red'
```

```
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


def main():
    root=Tk()
    #set the size of our window
    root.geometry('800x600')
    #Add a title to our window
    root.title('Awesome Gui -- It is way COOL')
    app = Application(master=root)

    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Exercise 10.1

(U) We have a lovely grid with labels. We don't want to disturb our grid, so let's put a new `Frame` on top. Notice how our frame lines up on the grid. It makes it really easy to see how setting the `row` and `column` alignment works and also the `rowspan` and `columnspan`.

Doc ID: 6689695

```python
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #----------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()
        #----------------------------------

        #--- Add Frame 1 ---
        self.add_frame1()
        #--------------------


    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #-------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
        #-------------------------------------------------------------

    def colorgen(self):
```

```
        '''
        Generator function that alternates between red and blue
        '''
        while True:
            yield 'red'
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


    def add_frame1(self):
        '''
        Add a frame with a text area to put stuff in.
        '''

        self.frame1=Frame(self,bg='red')
        self.frame1.grid(row=0,column=6,rowspan=10,columnspan=10,stick=self.ALL)


def main():
    root=Tk()
    root.geometry('800x600')
    root.title('Awesome Gui -- It is way COOL')
    app = Application(master=root)

    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Exercise 10.2

Great! Now let's put a widget in our frame. Notice that while you can't mix `grid` and `pack` inside a conatainer, we can use `pack` inside our `Frame` even though we were using `grid` for our top level window.

Exercise for reader: our Frame was red, but now that we added a text widget it doesn't look red anymore. What happened!? If we actually wanted a red text widget, what should we do differently?

Doc ID: 6689695

```python
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #----------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()
        #----------------------------------

        #----Add Frame 1-------
        self.add_frame1()

    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #-------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
        #-------------------------------------------------------------

    def colorgen(self):
        '''
        Generator function that alternates between red and blue
```

```
        '''
        while True:
            yield 'red'
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


    def add_frame1(self):
        '''
        A frame is a nice way to show how to map out a grid
        '''

        self.frame1=Frame(self,bg='red')
        self.frame1.grid(row=0,column=6,rowspan=10,columnspan=10,stick=self.ALL)
        self.frame1.text_w=Text(self.frame1)
        self.frame1.text_w.pack(expand=True,fill=BOTH)


def main():
    root=Tk()
    root.geometry('800x600')
    root.title('Awesome Gui -- It is way COOL')
    app = Application(master=root)

    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Exercise 10.3

Let's add some another frame.

Doc ID: 6689695

```python
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #----------------------------------

        #--- Make a checkerboard with Labels of different colors
        self.checkers_main_win()
        #----------------------------------

        #----Add Frame 1-------
        self.add_frame1()

        #---- Add Frame 2 -----
        self.add_frame2()

    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #-------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
        #-------------------------------------------------------------
```

```
    def colorgen(self):
        '''
        Generator function that alternates between red and blue
        '''
        while True:
            yield 'red'
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


    def add_frame1(self):
        '''
        Add a frame with a text area to put stuff in.
        '''

        self.frame1=Frame(self,bg='red')
        self.frame1.grid(row=0,column=6,rowspan=10,columnspan=10,stick=self.ALL)
        self.frame1.text_w=Text(self.frame1)
        self.frame1.text_w.pack(expand=True,fill=BOTH)

    def add_frame2(self):
        # Green frame!
        self.frame2 = Frame(self,bg='green')
        self.frame2.grid(row=0,column=0,rowspan=5,columnspan=6,sticky=self.ALL)



def main():
    root=Tk()
    root.geometry('800x600')
    root.title('Awesome Gui -- It is way COOL')
    app = Application(master=root)

    app.mainloop()

if __name__ == '__main__':
    main()
```

What if we wanted our GUI to do something when we click on our green frame? `Frame` doesn't allow us to set `command`. But wait! Hope is not lost! We can bind our frame to an event...

Binding to an event

(U) Not all widgets have a `command` option, but that doesn't mean you can't interact with them. Also, there may be instances when you want a response from the user other than a mouse click (which is what the built in `command` function responds to). In these instances, you want to bind your widget to the appropriate event.

Format: modifier (optional) - event type - detail (optional)
For example: `<Button-1>` modifier is `none` , event type is `Button` and detail is one. This means the event is the left mouse button was clicked. For the right mouse button, you would use two as your detail.

Common event types: `Button` , `ButtonRelease` , `KeyRelease` , `Keypress` , `FocusIn` , `FocusOut` , `Leave` (when the mouse leaves the widget), and `MouseWheel` .

Common modifiers: `Alt` , `Any` (used like `<Any-KeyPress>`), `Control` , `Double` (used like `<Double-Button-1>`)

Common details: These will vary widely based on the event type. Most commonly you will specify the key for `KeyPress` , ex: `<KeyPress-F1>`

Exercise 10.4

Let's bind our green frame to the left mouse button click and print out the coordinates of the click. Since printing to our notebook or the command line is not terribly useful for GUI's let's also display the coordinates in our text field

Doc ID: 6689695

```python
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #----------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()
        #----------------------------------

        #----Add Frame 1-------
        self.add_frame1()

        #---- Add Buttons -----
        self.add_frame2()

    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #-------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
        #-------------------------------------------------------------
```

```
    def colorgen(self):
        '''
        Generator function that alternates between red and blue
        '''
        while True:
            yield 'red'
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


    def add_frame1(self):
        '''
        Add a frame with a text area to put stuff in.
        '''

        self.frame1=Frame(self,bg='red')
        self.frame1.grid(row=0,column=6,rowspan=10,columnspan=10,stick=self.ALL)
        self.frame1.text_w=Text(self.frame1)
        self.frame1.text_w.pack(expand=True,fill=BOTH)

    def add_frame2(self):
        # Green frame!
        self.frame2 = Frame(self,bg='green')
        self.frame2.grid(row=0,column=0,rowspan=5,columnspan=6,sticky=self.ALL)
        self.frame2.bind('<Button-1>',self.frame2_handler)

    def frame2_handler(self, event):
        '''Handles events from frame2'''
        msg = 'Frame 2 clicked at {} {}'.format(event.x,event.y)
        print(msg)
        self.frame1.text_w.delete(1.0,END)
        self.frame1.text_w.insert(END, msg)

def main():
    root=Tk()
    root.geometry('800x600')
    root.title('Awesome Gui -- It is way COOL')
    app = Application(master=root)

    app.mainloop()
```

```
if __name__ == '__main__':
    main()
```

(U) Example 10.5

Great! Let's add some buttons now. Even though buttons have a `command` attribute, it can be tricky to pass information about the button being clicked using it. We would have to write a separate function for each button! Instead, let's use a key binding so we can access the button `text` from the event. Notice when we arrange the buttons we have to allign them by incriments of three since they span three columns.

Doc ID: 6689695

```python
from tkinter import *

class Application(Frame):
    '''
    Application -- the main app for the Frame... It all happens here.

    '''
    # Let's define some Class attributes
    mainwin_rows=11
    mainwin_cols=15
    ALL=N+S+E+W

    def __init__(self,master=None):
        '''
        Constructor
        '''
        # Call Frames Constructor
        Frame.__init__(self,master)

        # -- Call Private Grid Layout method
        self.__conf_self_grid_size()
        #----------------------------------

        #--- Make a checkerboard with labels of different colors
        self.checkers_main_win()
        #----------------------------------

        #----Add Frame 1-------
        self.add_frame1()

        #----Add Frame 2-------
        self.add_frame2()
        #---- Add Buttons -----
        self.add_buttons()

    def __conf_self_grid_size(self):
        '''
        I'm laying out the master grid
        '''
        # These next two lines ensure that the grid takes up the entire
        # window
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0,weight=1)
        #-------------------------------------------------------------
        #--- Now creating a grid on the main window
        for i in range(self.mainwin_rows):
            self.rowconfigure(i, weight=1)
        for j in range(self.mainwin_cols):
            self.columnconfigure(j, weight=1)
        self.grid(sticky=self.ALL)
```

```
    #-------------------------------------------------------------

    def colorgen(self):
        '''
        Generator function that alternates between red and blue
        '''
        while True:
            yield 'red'
            yield 'blue'

    def checkers_main_win(self):
        '''
        Creates a checkerboard pattern grid layout
        '''
        colors = self.colorgen()

        for r in range(self.mainwin_rows):
            for c in range(self.mainwin_cols):
                txt = 'Item {0}, {1}'.format(r,c)
                l = Label(self,text=txt,bg=next(colors))
                l.grid(row=r,column=c,sticky=self.ALL)


    def add_frame1(self):
        '''
        Add a frame with a text area to put stuff in.
        '''

        self.frame1=Frame(self,bg='red')
        self.frame1.grid(row=0,column=6,rowspan=10,columnspan=10,stick=self.ALL)
        self.frame1.text_w=Text(self.frame1)
        self.frame1.text_w.pack(expand=True,fill=BOTH)

    def add_frame2(self):
        # Green frame!
        self.frame2 = Frame(self,bg='green')
        self.frame2.grid(row=0,column=0,rowspan=5,columnspan=6,sticky=self.ALL)
        self.frame2.bind('<Button-1>',self.frame2_handler)

    def frame2_handler(self, event):
        '''Handles events from frame2'''
        msg = 'Frame 2 clicked at {} {}'.format(event.x,event.y)
        print(msg)
        self.frame1.text_w.delete(1.0,END)
        self.frame1.text_w.insert(END, msg)


    def add_buttons(self):
        ''' Add buttons to the bottom'''
        self.button_list=[]
        button_labels = ['Red','Blue','Green','Black','yellow']
```

```
        for c,bt in enumerate(button_labels):
            b = Button(self,text=bt)
            #span three columns and set the column alignment as
            #multiples of three.
            b.grid(row=10,column=c*3,columnspan=3,sticky=self.ALL)
            #Bind buttons to button click.
            b.bind('<Button-1>',self.buttons_handler)
            self.button_list.append(b)

    def buttons_handler(self,event):
        '''
        Event Handler for the buttons
        '''
        button_clicked = event.widget['text']
        print(button_clicked)

        self.frame1.text_w.delete(1.0,END)
        self.frame1.text_w.insert(END, button_clicked)

def main():
    root=Tk()
    root.geometry('800x600')
    root.title('Awesome Gui -- It is way COOL')
    app = Application(master=root)

    app.mainloop()

if __name__ == '__main__':
    main()
```

(U) Dialogs

(U) `Tkinter` provides a collection of ready-made dialog boxes to use:

- `showinfo`
- `showwarning`
- `showerror`
- `askquestion`
- `askokcancel`
- `askyesno`
- `askyesnocancel`
- `askretry_cancel`

```
from tkinter import messagebox as mBox
from tkinter import Tk
root = Tk()
#This keeps the top level window from being drawn
root.withdraw()

mBox.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2016.')
#mBox.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:\nWarning: There might be a bug in this code.')
#mBox.showerror('Python Message Error Box', 'A Python GUI created using tkinter:\nError: Houston ~ we DO have a serious PROBLEM!')
```

Useful References:

Python GUI Programming Cookbook

By Burkhard A. Meier

On safari: http://ncmd-ebooks-1.ncmd.nsa.ic.gov/9781785283758

Great overview of different widgets and also using "themed" `tk` widgets or `ttk` widgets.

Tkinter GUI Application Development Blueprints

By: Bhaskar Chaudhary

On safari: https://ncmd-ebooks-1.ncmd.nsa.ic.gov/9781785889738

Not all of the packages used in the examples are available, but this book is still a great reference for how to build and structure larger GUI projects.

Programming Python, 4th Edition

By: Mark Lutz

On safari: https://ncmd-ebooks-1.ncmd.nsa.ic.gov/9781449398712

Not a dedicated GUI book, this book does have several chapters cover different GUI aspects. It spends more time explaining the underlying logic of GUI's and covering special cases that can trip you up. It starts from the basics, but moves quickly, so I might not be the best resource for total novices.

Doc ID: 6689695

Python GUI Programming Cookbook

0

(b)(3)-P.L. 86-36

Updated over 1 year ago by [redacted]

3 4 385 98

tk gui python fcs61 tkinter safari

(U) Code from "Python GUI Programming Cookbook"

Recommendations

Python GUI Programming Cookbook

By Burkhard A. Meier

On safari: http://ncmd-ebooks-1.ncmd.nsa.ic.gov/9781785283758

Doc ID: 6689695

```python
#=====================================================================
#Imports
#=====================================================================
import tkinter as tk                    # basic gui
from tkinter import ttk                 # "themed tk" improved gui options
from tkinter import scrolledtext        # For scrolling text boxes
from tkinter import Menu                # For creating menus
from tkinter import Spinbox             # For creating spin boxes
from tkinter import messagebox as mBox  # For message boxes

#=====================================================================
# Initial framework setup for window, tabs, frames, etc.
#=====================================================================

win = tk.Tk()                           # Create instance
win.title("Python GUI")                 # Add a title
win.iconbitmap(r'U:\private\anaconda3\DLLs\pyc.ico')  #Change icon

tabControl = ttk.Notebook(win)          # Create Tab Control

tab1 = ttk.Frame(tabControl)            # Create a tab
tabControl.add(tab1, text='Tab 1')      # Add the tab

tab2 = ttk.Frame(tabControl)            # Add a second tab
tabControl.add(tab2, text='Tab 2')      # Make second tab visible

tab3 = ttk.Frame(tabControl)            # Add a third tab
tabControl.add(tab3, text='Tab 3')      # Make second tab visible

tabControl.pack(expand=1, fill="both")  # Pack to make visible

# We are creating a container frame to hold all other widgets in tab1
monty = ttk.LabelFrame(tab1, text=' Monty Python ')
monty.grid(column=0, row=0, padx=8, pady=4)


# We are creating a container frame to hold all other widgets in tab2
monty2 = ttk.LabelFrame(tab2, text=' The Snake ')
monty2.grid(column=0, row=0, padx=8, pady=4)
```

```
#======================================================================
#Callback functions
#======================================================================

def clickMe():                          # Function for when button is clicked
    action.configure(text='Hello ' + name.get()+' number '+numberChosen.get()+'! ')

# Set Radiobutton global variables into a list.
colors = ["Pink", "Magenta", "Purple"]

#We have also changed the callback function to be zero-based, using the list instead of module-level global variables.
# Radiobutton callback function
def radCall():
   radSel=radVar.get()
   if   radSel == 0: monty2.configure(text=colors[0])
   elif radSel == 1: monty2.configure(text=colors[1])
   elif radSel == 2: monty2.configure(text=colors[2])

def _quit():
    win.quit()
    win.destroy()
    #exit()

# Display a Message Box
# Callback function
def _msgBox():
    #mBox.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2016.')
    #mBox.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:\nWarning: There might be a bug in this
    #mBox.showerror('Python Message Error Box', 'A Python GUI created using tkinter:\nError: Houston ~ we DO have a serious PRO
    answer = mBox.askyesno("Python Message Dual Choice Box", "Are you sure you really wish to do this?")
    print(answer)

# Spinbox callback
def _spin():
    value = spin.get()
    print(value)
    scr.insert(tk.INSERT, value + '\n')

#======================================================================
# Create Menu bar
```

Doc ID: 6689695

```
#=====================================================================
menuBar = Menu(win)                                          #create menu bar
win.config(menu=menuBar)                                     #add menu bar to gui

fileMenu = Menu(menuBar, tearoff = 0)                        #create menu
fileMenu.add_command(label="New")                            #add option to menu
fileMenu.add_separator()                                     #add separator to menu
fileMenu.add_command(label="Exit", command=_quit)            #add option to menu
menuBar.add_cascade(label="File", menu=fileMenu)             #add menu to menu bar

# Add another Menu to the Menu Bar and an item
helpMenu = Menu(menuBar, tearoff=0)                          #create second menu
helpMenu.add_command(label="About", command=_msgBox)         #add menu item
menuBar.add_cascade(label="Help", menu=helpMenu)             #add menu to menu bar

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Contents Tab1
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#=====================================================================
# Create labels
#=====================================================================

ttk.Label(monty, text="Enter a name:").grid(column=0, row=0,sticky=tk.W)          # text label position (0,0)
ttk.Label(monty, text="Choose a number:").grid(column=1, row=0, sticky=tk.W)       # Another label (with location)

#=====================================================================
# Text entry box
#=====================================================================

# Adding a Textbox Entry widget
name = tk.StringVar()                                        # tk's version of a string (storage variable)
nameEntered = ttk.Entry(monty, width=12, textvariable=name)  # Entry box
nameEntered.grid(column=0, row=1,sticky=tk.W)                # Entry box location
nameEntered.focus()                                          # when app starts, put curser in box

#=====================================================================
# Combo box
#=====================================================================
number = tk.StringVar()                                      # tk string variable to hold numbewr
numberChosen = ttk.Combobox(monty, width=12, textvariable=number, state = 'readonly') #combo box
```

```
numberChosen['values'] = (1, 2, 4, 42, 100)                    # options for combo box
numberChosen.grid(column=1, row=1,sticky=tk.W)                 # Location combo box
numberChosen.current(0)


#============================================================================
# Button
#============================================================================
action = ttk.Button(monty, text="Click Me!", command=clickMe)    # create button with text and function for when clicked
action.grid(column=2, row=1,sticky=tk.W)                         # Position Button in second row, second column (zero-based)

#============================================================================
# Spinbox
#============================================================================

# Adding a Spinbox widget
spin = Spinbox(monty, values=(1, 2, 4, 42, 100), width=5, bd=8, command=_spin)          #spinbox set of values
spin.grid(column=0, row=2)

# Adding a second Spinbox widget
spin2 = Spinbox(monty, from_=0, to=10, width=5, bd=8, relief = tk.RIDGE, command=_spin)  #spinbox range of values

#Alternate relief options:
#spin2 = Spinbox(monty, values=(0, 50, 100), width=5, bd=20, relief = tk.FLAT, command=_spin)
#spin2 = Spinbox(monty, values=(0, 50, 100), width=5, bd=20, relief = tk.GROOVE, command=_spin)

spin2.grid(column=1, row=2)

#============================================================================
# Checkboxes
#============================================================================

# Creating three checkbuttons
chVarDis = tk.IntVar()                                         # tk int variable, for check box state
check1 = tk.Checkbutton(monty, text="Disabled", variable=chVarDis, state='disabled') #disabled checkbox
check1.select()                                                # add checkmark
check1.grid(column=0, row=4, sticky=tk.W)                      # position checkbox. sticky=tk.W means aligned west

chVarUn = tk.IntVar()                                          # another tk in variable for checkbox state
check2 = tk.Checkbutton(monty, text="UnChecked", variable=chVarUn)                  #un-checked checkbox
check2.deselect()                                              # set checkbox to not checked
```

```
check2.grid(column=1, row=4, sticky=tk.W)                    # position checkbox

chVarEn = tk.IntVar()                                        # checkbox int variable for checkbox state
check3 = tk.Checkbutton(monty, text="Enabled", variable=chVarEn)                    #Checked checkbox
check3.select()                                              # set checkbox to checked
check3.grid(column=2, row=4, sticky=tk.W)                    # position checkbox

#============================================================================
# Scrollbox
#============================================================================

# Using a scrolled Text control
scrolW  = 30                                                  # scrollbox width
scrolH  =  3                                                  # scrollbox height
scr = scrolledtext.ScrolledText(monty, width=scrolW, height=scrolH, wrap=tk.WORD)      # create scroll box
scr.grid(column=0, row = 5, columnspan=3,sticky='WE')        # position scroll box

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Contents Tab2
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#============================================================================
# Radio Buttons
#============================================================================

# create three Radiobuttons using one variable
radVar = tk.IntVar()

#Next we are selecting a non-existing index value for radVar.
radVar.set(99)

#Now we are creating all three Radiobutton widgets within one loop.
for col in range(3):
    curRad = 'rad' + str(col)
    curRad = tk.Radiobutton(monty2, text=colors[col], variable=radVar, value=col, command=radCall)
    curRad.grid(column=col, row=6, sticky=tk.W)

#============================================================================
# Labels in a labelsframe
#==================================================================
# Create a container to hold labels
```

```
labelsFrame = ttk.LabelFrame(monty2, text=' Labels in a Frame ')
labelsFrame.grid(column=0, row=7)                # position with padding

# Place labels into the container element
ttk.Label(labelsFrame, text="Label1").grid(column=0, row=0)
ttk.Label(labelsFrame, text="Label2").grid(column=0, row=1)
ttk.Label(labelsFrame, text="Label3").grid(column=0, row=2)

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Contents Tab3
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

tab3 = tk.Frame(tab3, bg='purple')
tab3.pack()


#=======================================================================
#Callback functions
#=======================================================================

def checked():

    if chVarCr.get():

        canvas = tk.Canvas(tab3, width=150, height=80, highlightthickness=0, bg='blue')
        canvas.grid(row=1, column=0)
        canvas = tk.Canvas(tab3, width=150, height=80, highlightthickness=0, bg='blue')
        canvas.grid(row=0, column=1)
    else:
        canvas = tk.Canvas(tab3, width=150, height=80, highlightthickness=0, bg='green')
        canvas.grid(row=1, column=0)
        canvas = tk.Canvas(tab3, width=150, height=80, highlightthickness=0, bg='green')
        canvas.grid(row=0, column=1)

#=======================================================================
# Checkbox and canvases
#=======================================================================


for pinkColor in range(0,2):
    canvas = tk.Canvas(tab3, width=150, height=80, highlightthickness=0, bg='pink')
```

```
    canvas.grid(row=pinkColor, column=pinkColor)
chVarCr = tk.IntVar()                                    # another tk in variable for checkbox state
checkCr = tk.Checkbutton(tab3, text="Color", variable=chVarCr,command=checked)          #un-checked checkbox
checkCr.deselect()                                       # set checkbox to not checked
checkCr.grid(row=0,column=0)                     # position checkbox



#======================================================================
# Display GUI
#======================================================================
win.mainloop()
```

Examples of other message boxes

```
from tkinter import messagebox as mBox
from tkinter import Tk
root = Tk()
root.withdraw()

mBox.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2016.')
#mBox.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:\nWarning: There might be a bug in this coc
#mBox.showerror('Python Message Error Box', 'A Python GUI created using tkinter:\nError: Houston ~ we DO have a serious PROBLEM
```

```
from tkinter import *
root = Tk()
root.withdraw()
messagebox.showerror('Python Message Error Box', 'A Python GUI created using tkinter:\nError: Houston ~ we DO have a serious PR
```

```
from tkinter import messagebox as mBox
help(mBox)
```

Module: Logging

(b)(3)-P.L. 86-36

0

Updated over 2 years ago by [redacted] in COMP 3321

3 4 152 46

fcs6 access extra python

(U) Module: Logging

Recommendations

(U) There comes a time in every developer's career when he or she decides that there must be something better than debugging with `print` statements. Fortunately, it only takes about one minute to set up basic logging in Python, which can help you diagnose problems in your program during development, then suppress all that output when you use the program for real. After that, there's no looking back; you might even start to use a real debugger someday. The **logging** module has many advanced configuration options, but you'll probably see benefits even if you don't ever use any of them.

(U) At its heart, logging is a name for the practice of capturing and storing data and events from a program that are not part of the main output stream. Logging also frequently uses the concept of *severity levels*: some captured data indicates serious problems, other data provides useful information, and some details are useful only when trying to track down bugs in the logic. Capturing messages from any or all of the different severity levels depends on who invoked the program, along with how and why it's being run.

(U) The Basics

(U) The **logging** module is included in the standard library. To begin using it with the absolute minimum effort possible, import it and start writing messages at different levels.

```
import logging

logging.warning('You have been warned')

logging.critical('ABORT ABORT ABORT')

logging.info('This is some helpful information')
```