Homework 6

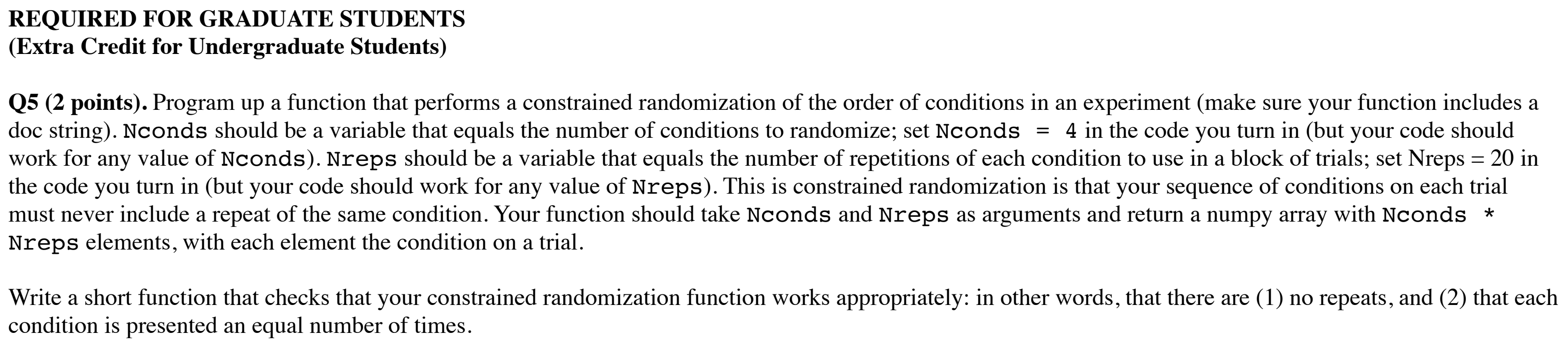
due Wed Oct 26 at start of class

`Homework6.pdf`

Homework 6

NOTE that I have added a required question for graduate students (extra credit for undergraduates)

`Homework6.pdf`



REQUIRED FOR GRADUATE STUDENTS
(Extra Credit for Undergraduate Students)

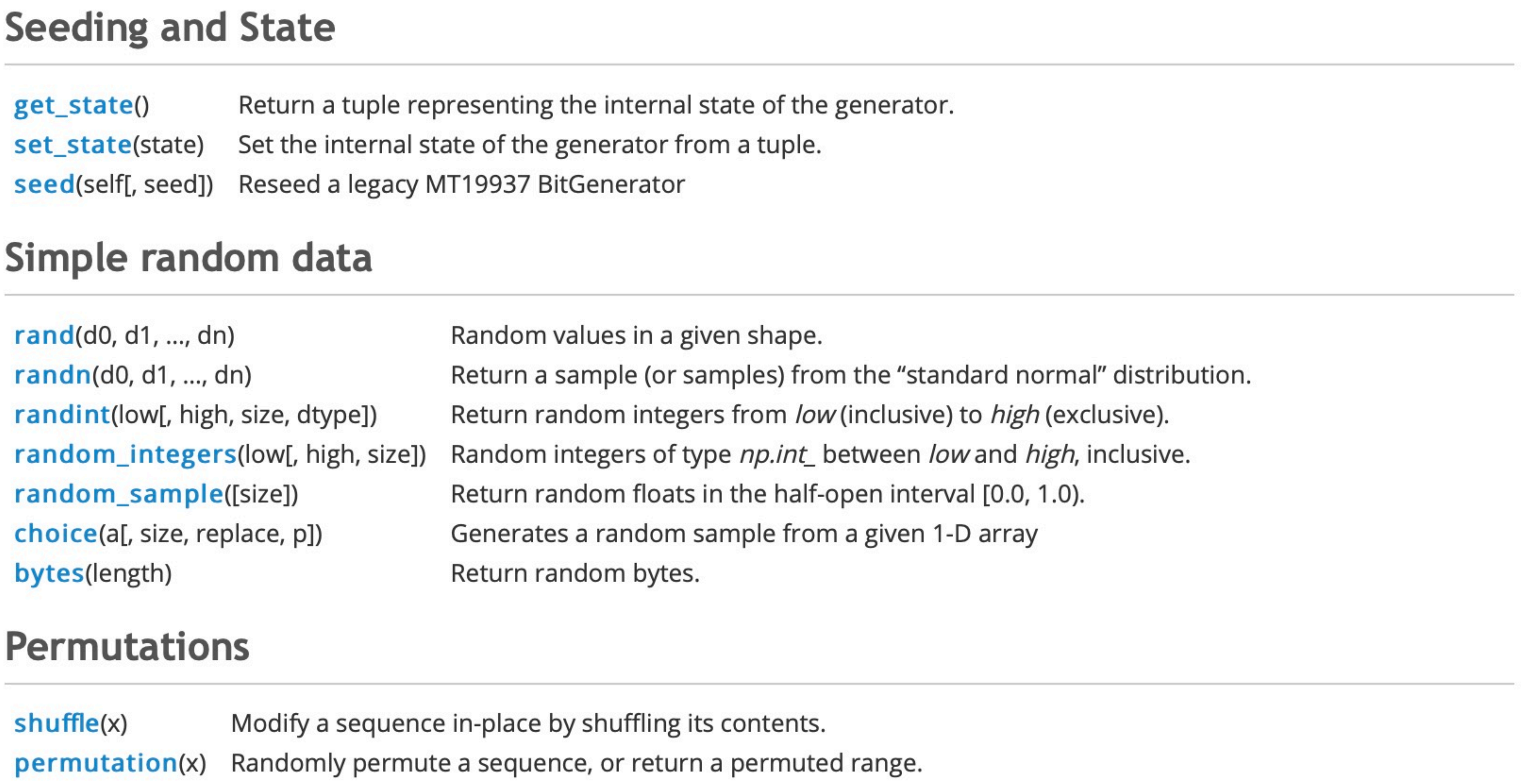
Q5 (2 points). Program up a function that performs a constrained randomization of the order of conditions in an experiment (make sure your function includes a doc string). `Nconds` should be a variable that equals the number of conditions to randomize; set `Nconds = 4` in the code you turn in (but your code should work for any value of `Nconds`). `Nreps` should be a variable that equals the number of repetitions of each condition to use in a block of trials; set Nreps = 20 in the code you turn in (but your code should work for any value of `Nreps`). This is constrained randomization is that your sequence of conditions on each trial must never include a repeat of the same condition. Your function should take `Nconds` and `Nreps` as arguments and return a numpy array with `Nconds * Nreps` elements, with each element the condition on a trial.

Write a short function that checks that your constrained randomization function works appropriately: in other words, that there are (1) no repeats, and (2) that each condition is presented an equal number of times.

download from Brightspace

`Random.ipynb`

note that I've added to this since what
was uploaded last week



Seeding and State

get_state()	Return a tuple representing the internal state of the generator.
set_state(state)	Set the internal state of the generator from a tuple.
seed(self[, seed])	Reseed a legacy MT19937 BitGenerator

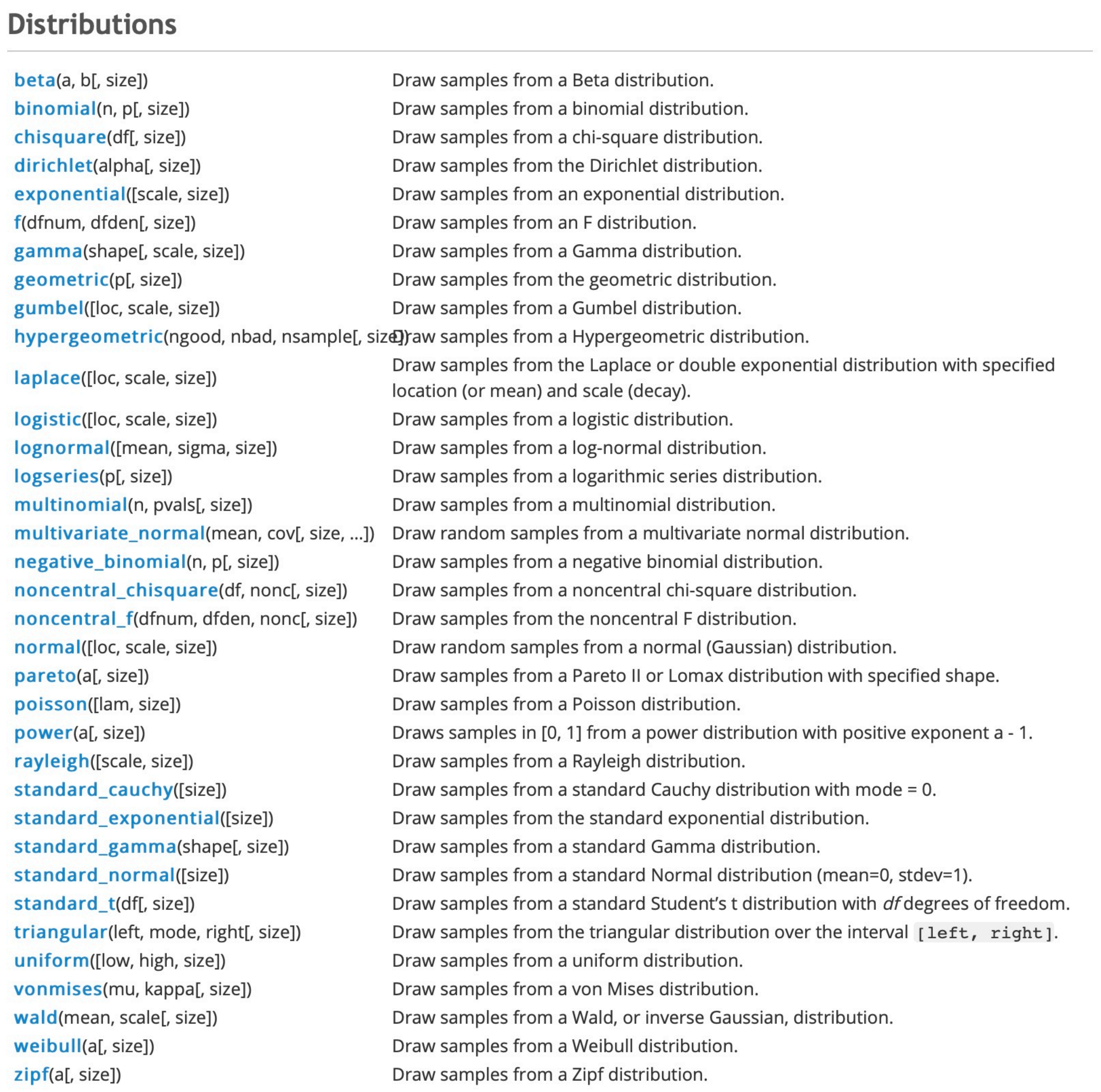
Simple random data

rand(d0, d1, ..., dn)	Random values in a given shape.
randn(d0, d1, ..., dn)	Return a sample (or samples) from the “standard normal” distribution.
randint(low[, high, size, dtype])	Return random integers from *low* (inclusive) to *high* (exclusive).
random_integers(low[, high, size])	Random integers of type *np.int_* between *low* and *high*, inclusive.
random_sample([size])	Return random floats in the half-open interval [0.0, 1.0).
choice(a[, size, replace, p])	Generates a random sample from a given 1-D array
bytes(length)	Return random bytes.

Permutations

shuffle(x)	Modify a sequence in-place by shuffling its contents.
permutation(x)	Randomly permute a sequence, or return a permuted range.

https://numpy.org/doc/stable/reference/random/legacy.html#numpy.random.RandomState



Distributions

beta(a, b[, size])	Draw samples from a Beta distribution.
binomial(n, p[, size])	Draw samples from a binomial distribution.
chisquare(df[, size])	Draw samples from a chi-square distribution.
dirichlet(alpha[, size])	Draw samples from the Dirichlet distribution.
exponential([scale, size])	Draw samples from an exponential distribution.
f(dfnum, dfden[, size])	Draw samples from an F distribution.
gamma(shape[, scale, size])	Draw samples from a Gamma distribution.
geometric(p[, size])	Draw samples from the geometric distribution.
gumbel([loc, scale, size])	Draw samples from a Gumbel distribution.
hypergeometric(ngood, nbad, nsample[, size])	Draw samples from a Hypergeometric distribution.
laplace([loc, scale, size])	Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).
logistic([loc, scale, size])	Draw samples from a logistic distribution.
lognormal([mean, sigma, size])	Draw samples from a log-normal distribution.
logseries(p[, size])	Draw samples from a logarithmic series distribution.
multinomial(n, pvals[, size])	Draw samples from a multinomial distribution.
multivariate_normal(mean, cov[, size, ...])	Draw random samples from a multivariate normal distribution.
negative_binomial(n, p[, size])	Draw samples from a negative binomial distribution.
noncentral_chisquare(df, nonc[, size])	Draw samples from a noncentral chi-square distribution.
noncentral_f(dfnum, dfden, nonc[, size])	Draw samples from the noncentral F distribution.
normal([loc, scale, size])	Draw random samples from a normal (Gaussian) distribution.
pareto(a[, size])	Draw samples from a Pareto II or Lomax distribution with specified shape.
poisson([lam, size])	Draw samples from a Poisson distribution.
power(a[, size])	Draws samples in [0, 1] from a power distribution with positive exponent a - 1.
rayleigh([scale, size])	Draw samples from a Rayleigh distribution.
standard_cauchy([size])	Draw samples from a standard Cauchy distribution with mode = 0.
standard_exponential([size])	Draw samples from the standard exponential distribution.
standard_gamma(shape[, size])	Draw samples from a standard Gamma distribution.
standard_normal([size])	Draw samples from a standard Normal distribution (mean=0, stdev=1).
standard_t(df[, size])	Draw samples from a standard Student's t distribution with *df* degrees of freedom.
triangular(left, mode, right[, size])	Draw samples from the triangular distribution over the interval `[left, right]`.
uniform([low, high, size])	Draw samples from a uniform distribution.
vonmises(mu, kappa[, size])	Draw samples from a von Mises distribution.
wald(mean, scale[, size])	Draw samples from a Wald, or inverse Gaussian, distribution.
weibull(a[, size])	Draw samples from a Weibull distribution.
zipf(a[, size])	Draw samples from a Zipf distribution.

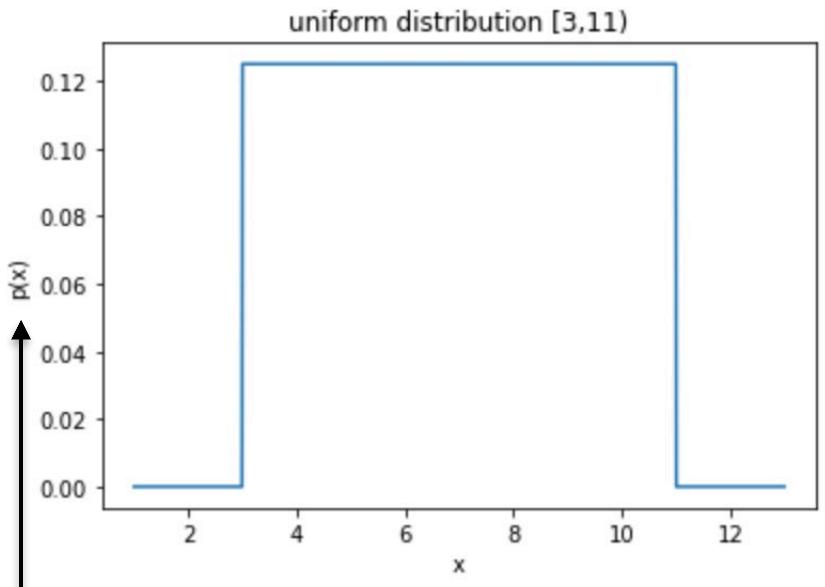
https://numpy.org/doc/stable/reference/random/legacy.html#numpy.random.RandomState

continuous discrete

probability density vs. probability mass function

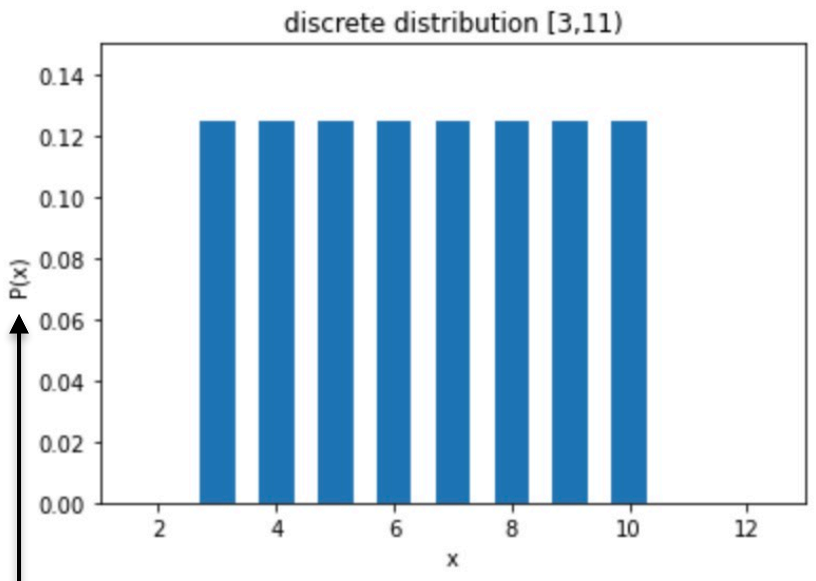
uniform distribution [a,b)

probability density function



discrete uniform distribution [a,b)

probability mass function



p(x) - likelihood

area under the curve (integral) equals 1.0

P(x) - probability

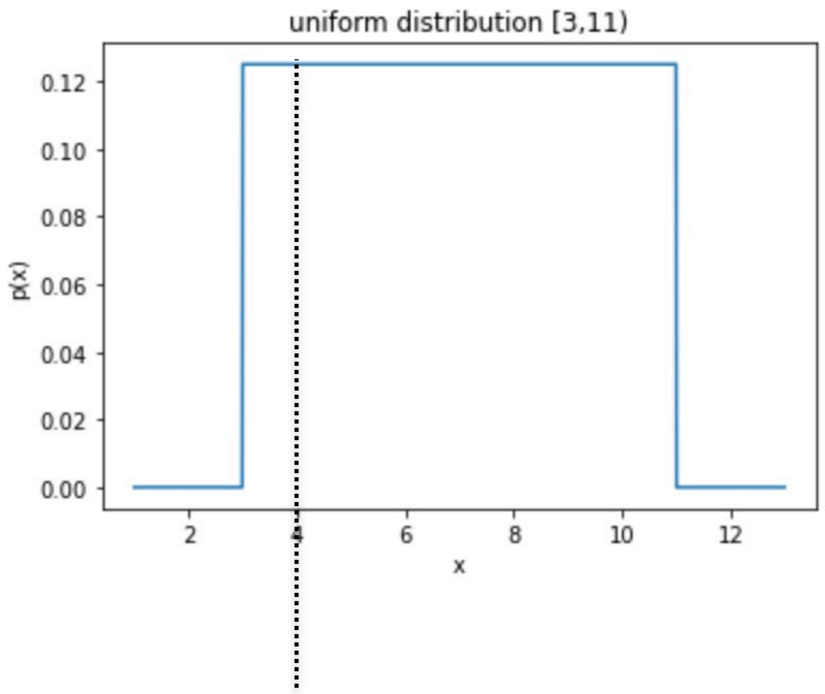
sum over all x values equals 1.0

continuous discrete

probability density vs. probability mass function

uniform distribution [a,b)

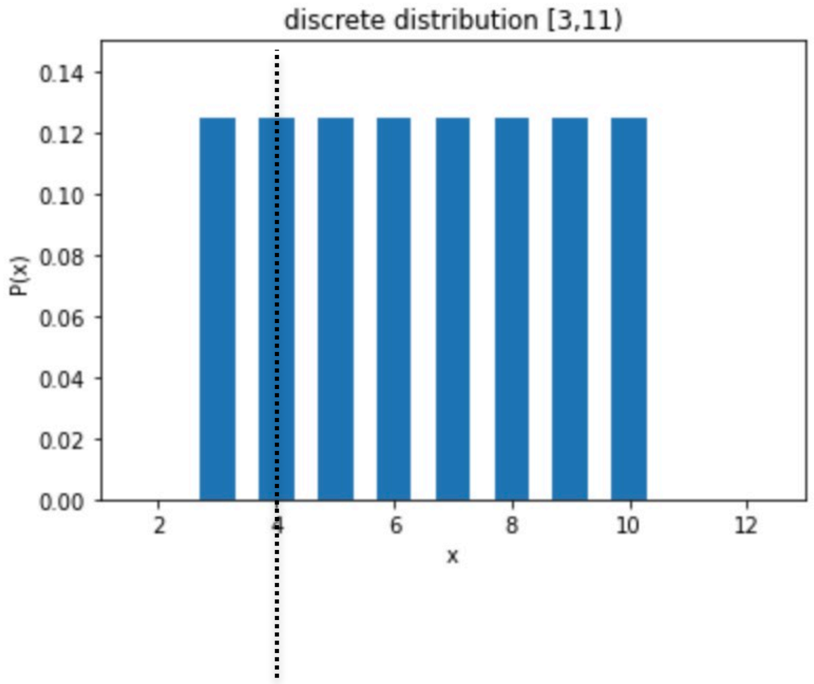
probability density function



P(x=4.0000000) = 0

discrete uniform distribution [a,b)

probability mass function



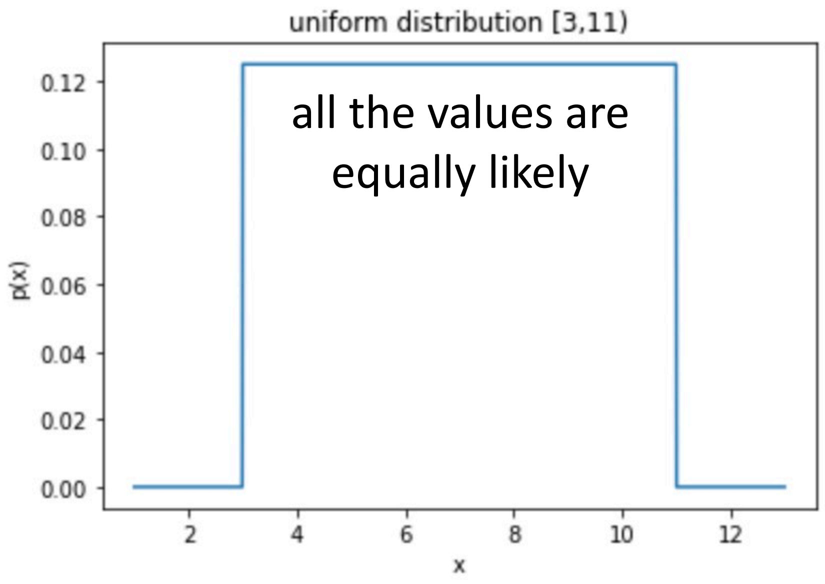
P(x=4) = 1/8

continuous discrete

probability density vs. probability mass function

uniform distribution [a,b)

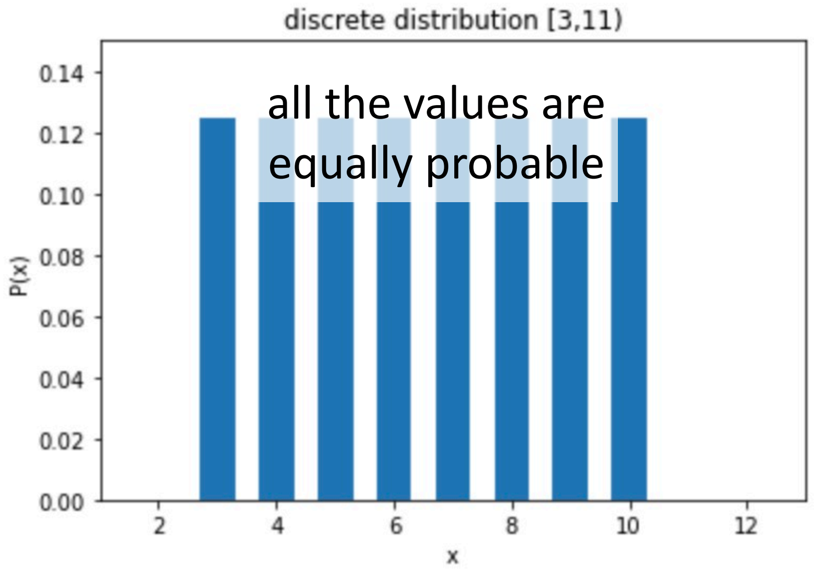
probability density function



all the values are equally likely

discrete uniform distribution [a,b)

probability mass function



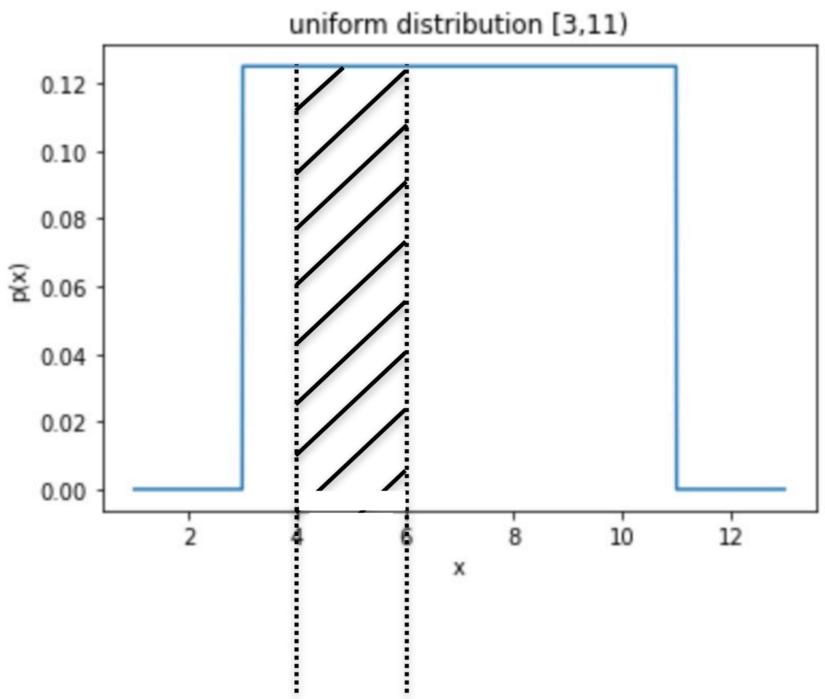
all the values are equally probable

continuous discrete

probability density vs. probability mass function

uniform distribution [a,b)

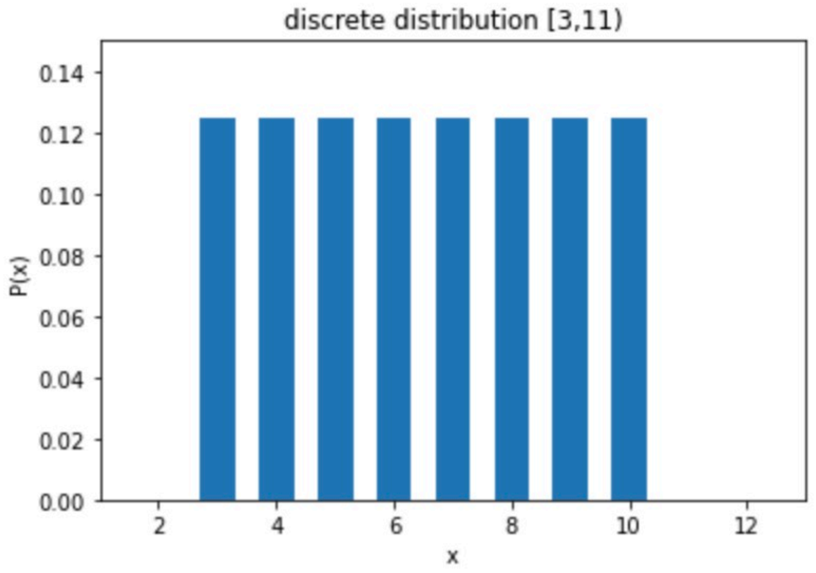
probability density function



P(4 ≤ x < 6) = .250

discrete uniform distribution [a,b)

probability mass function

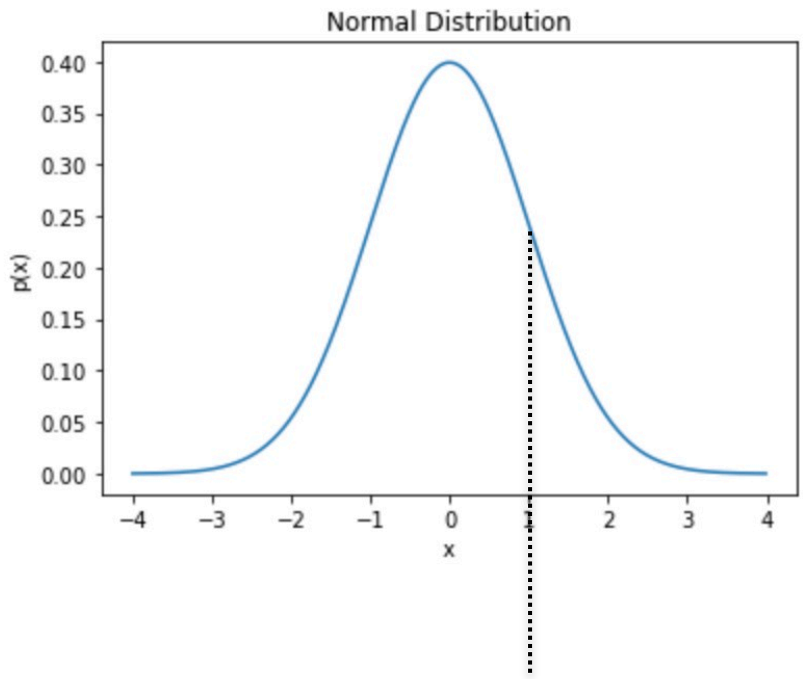


continuous

probability density

normal distribution (mean 0, stdev 1)

probability density function



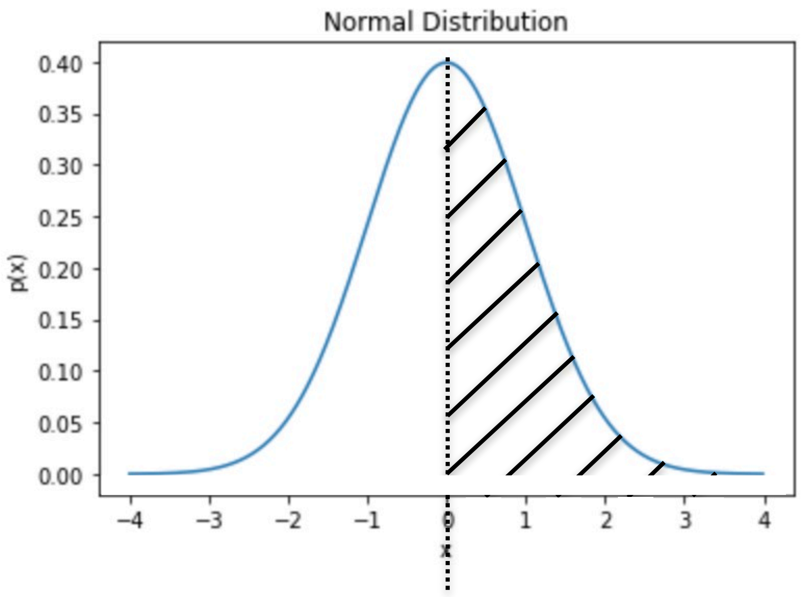
P(x=1.0000000) = 0

continuous

probability density

normal distribution (mean 0, stdev 1)

probability density function

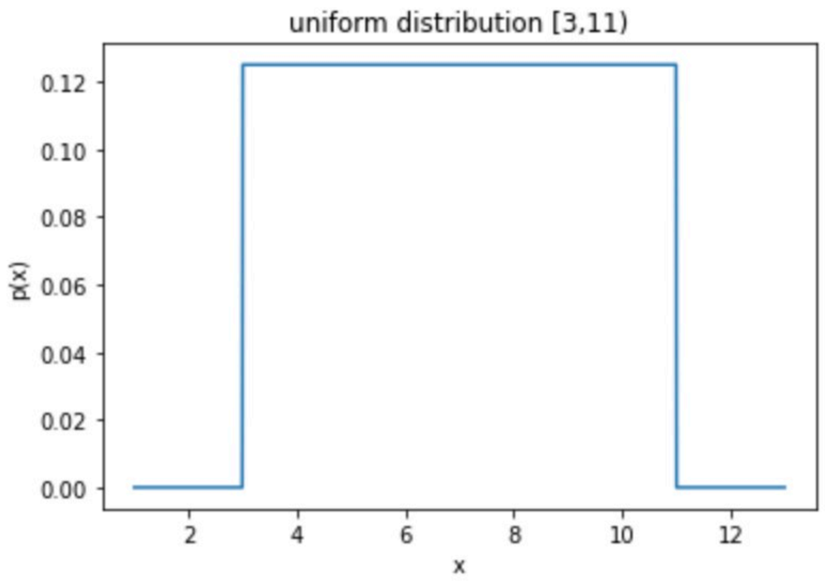


$P(x \geq 0) = 1/2$

probability density vs. cumulative distribution function

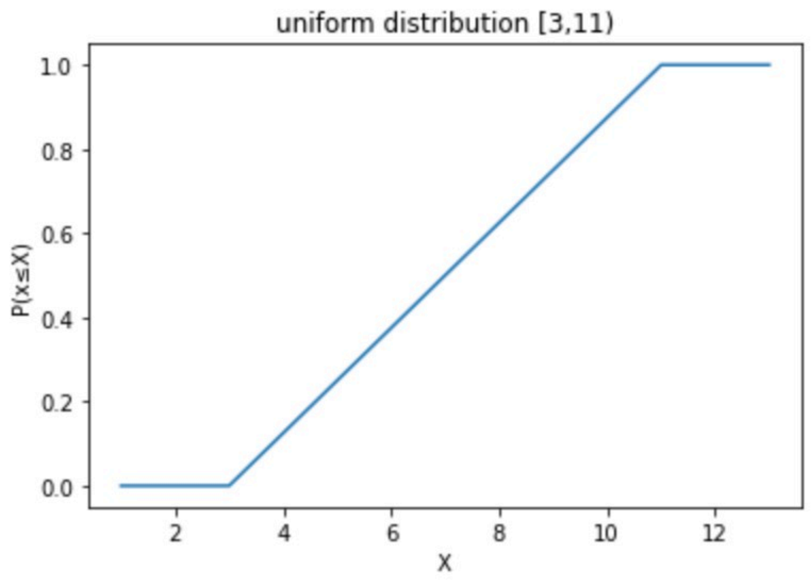
uniform distribution [a,b)

probability density function (pdf)



uniform distribution [a,b)

cumulative distribution function (cdf)



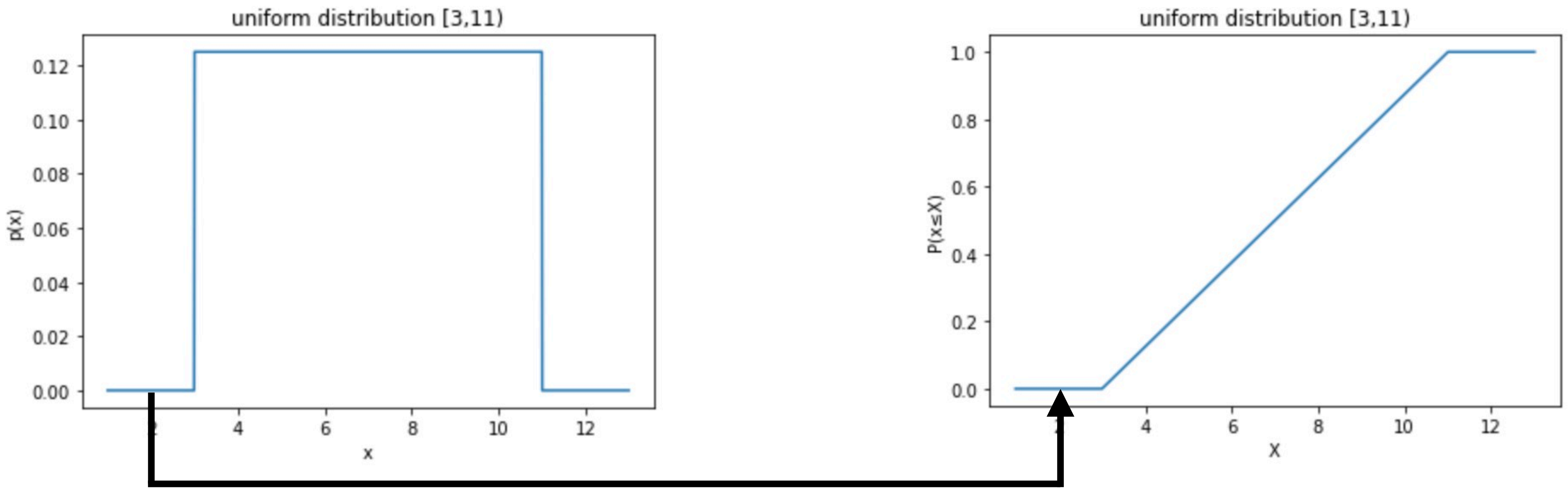
probability density vs. cumulative distribution function

uniform distribution [a,b)

probability density function (pdf)

uniform distribution [a,b)

cumulative distribution function (cdf)



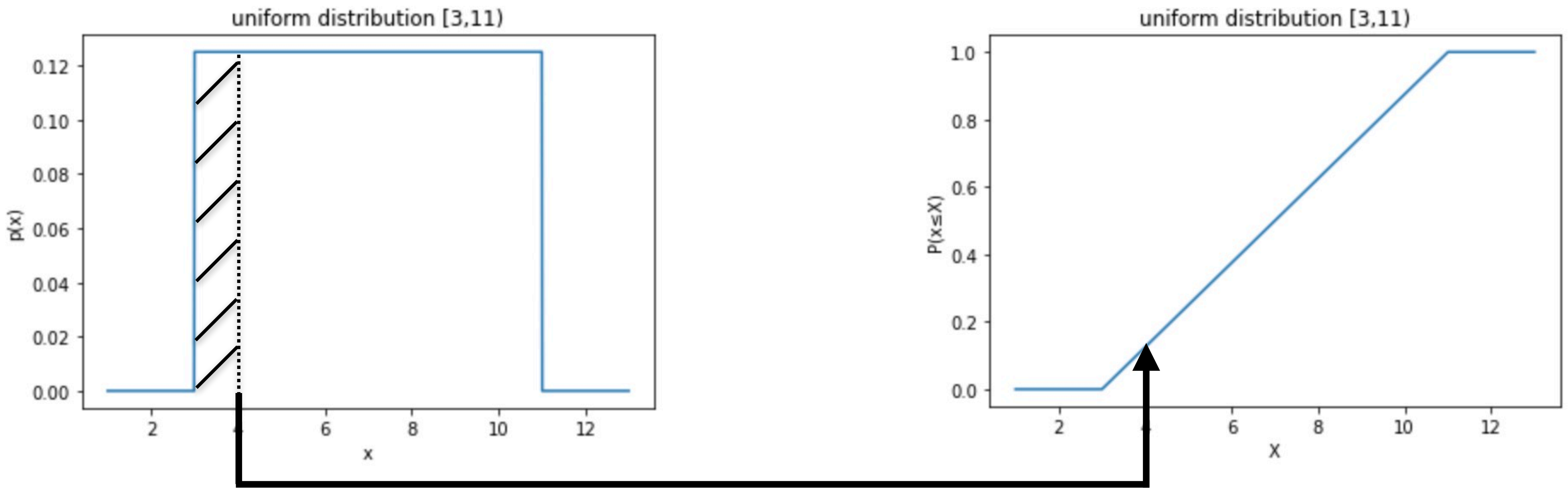
probability density vs. cumulative distribution function

uniform distribution [a,b)

probability density function (pdf)

uniform distribution [a,b)

cumulative distribution function (cdf)



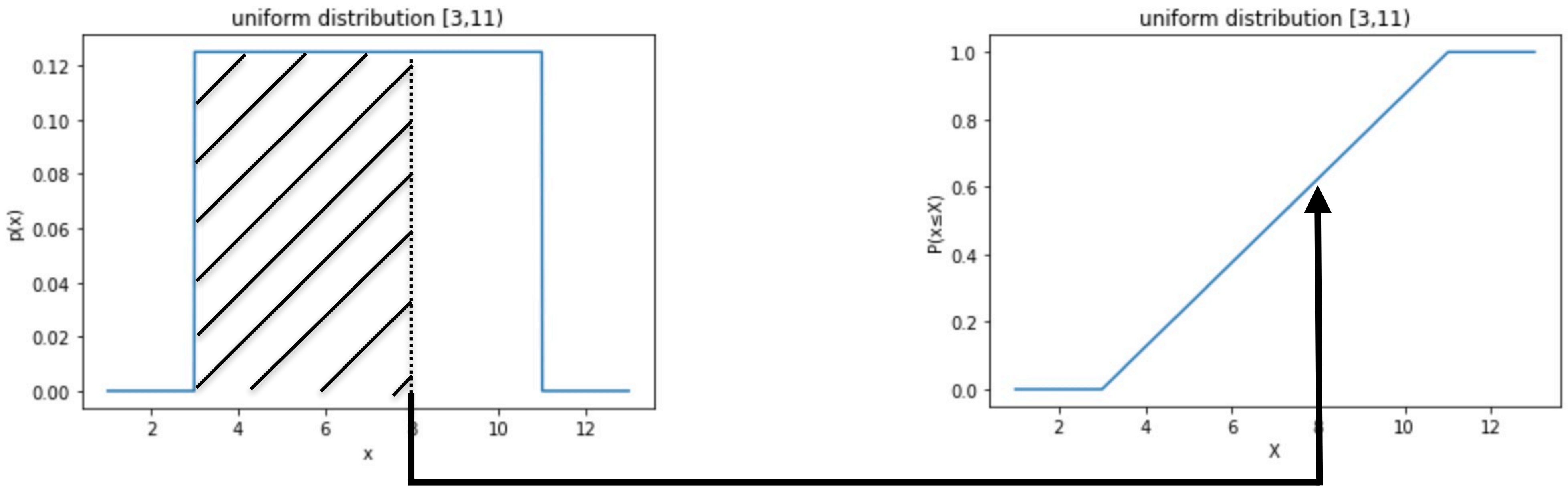
probability density vs. cumulative distribution function

uniform distribution [a,b)

probability density function (pdf)

uniform distribution [a,b)

cumulative distribution function (cdf)



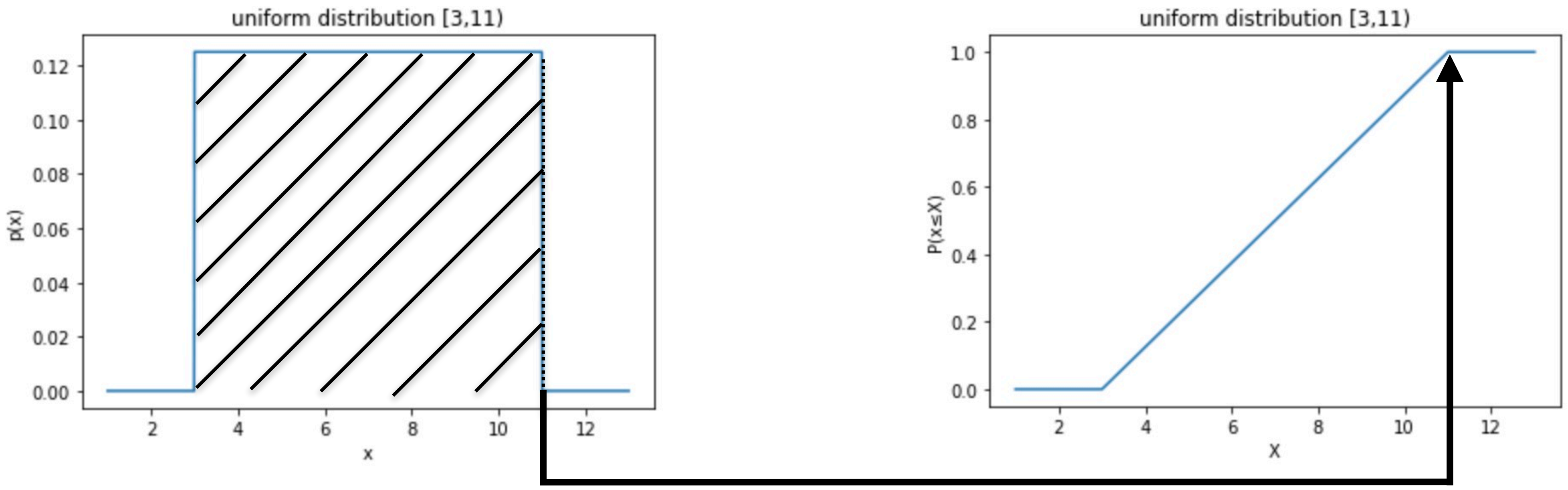
probability density vs. cumulative distribution function

uniform distribution [a,b)

probability density function (pdf)

uniform distribution [a,b)

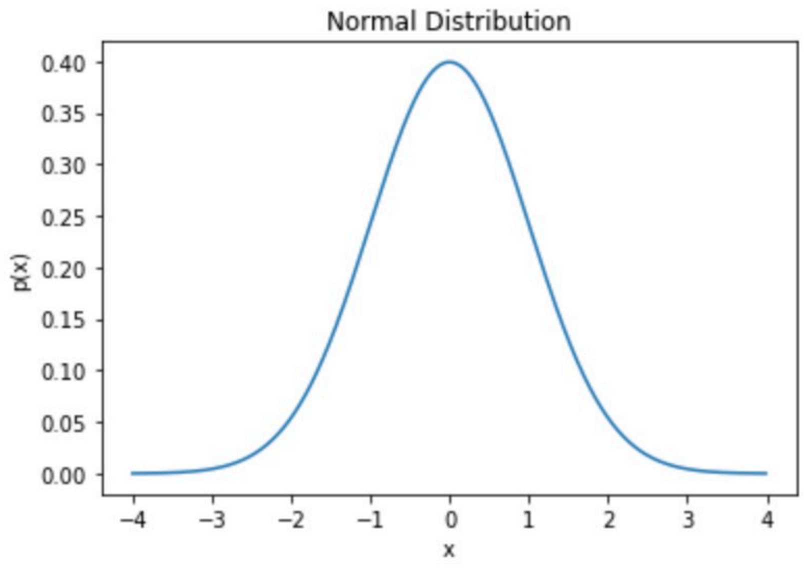
cumulative distribution function (cdf)



probability density vs. cumulative distribution function

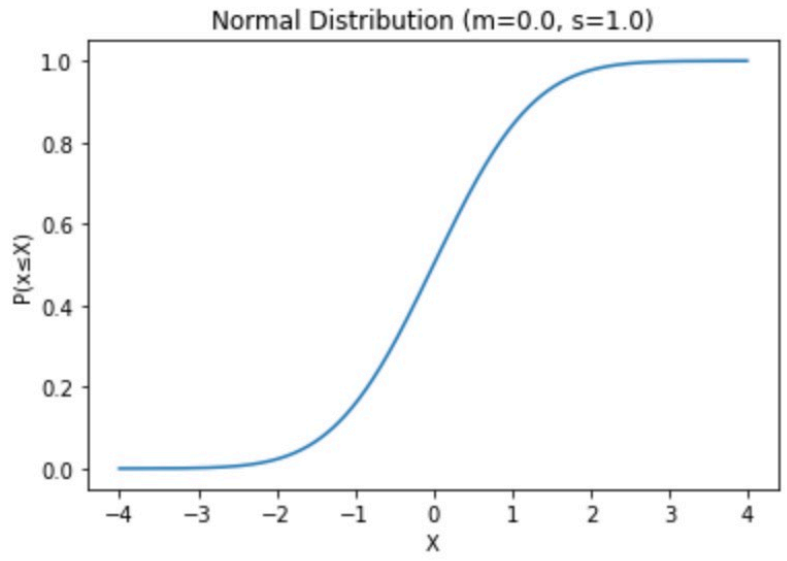
normal distribution (mean 0, stdev 1)

probability density function (pdf)



normal distribution (mean 0, stdev 1)

cumulative distribution function (cdf)



random sample from Bernoulli distribution

1 with probability `p` (Success)

0 with probability `1-p` (Failure)

weighted coin flip

https://en.wikipedia.org/wiki/Bernoulli_process

random sample from Bernoulli distribution

1 with probability `p` (Head)

0 with probability `1-p` (Tail)

weighted coin flip

https://en.wikipedia.org/wiki/Bernoulli_process

random sample from Bernoulli distribution

1 with probability `p` (Head)

0 with probability `1-p` (Tail)

weighted coin flip

```
rnd = R.uniform()
if (rnd < p):
    val = 1
else:
    val = 0
```

sample from Binomial Distribution

number of "successes" (with prob `p`) given `N` attempts

1 with probability `p` (Head)

0 with probability `1-p` (Tail)

weighted coin flip

```
p = .5; N = 1
```

(reduces to a Bernoulli when `N` = 1)

```
val = R.binomial(N, p)
```

https://en.wikipedia.org/wiki/Binomial_distribution

Example: simulating the gambler's fallacy

imagine you have the chance to bet on flips of a fair coin

you observe the following sequence of flips:

H T T T H H H H H H H H H H H H H

Question:
After observing this sequence, do you think that the chance of getting a head on the next flip is greater or less than 50:50?

“gambler’s fallacy”

Example: simulating the gambler's fallacy

first, simulating a series of simple (fair) coin flips

```
import numpy.random as R

iterations = 10000
nheads = 0; ntries = 0; p = 0.5
for idx in range(iterations):
    rnd = R.uniform()
    if rnd < p:
        nheads = nheads + 1
    ntries = ntries + 1
print(nheads/ntries)
```

Example: simulating the gambler's fallacy

first, simulating a series of simple (fair) coin flips

```
import numpy.random as R

iterations = 10000
nheads = 0; ntries = 0; p = 0.5
for idx in range(iterations):
    rnd = R.uniform()
    if rnd < p:
        nheads = nheads + 1
    ntries = ntries + 1
print(nheads/ntries)
```

number of heads on `N` flips with prob `p` is a random sample from a binomial distribution; here, we're simulating this

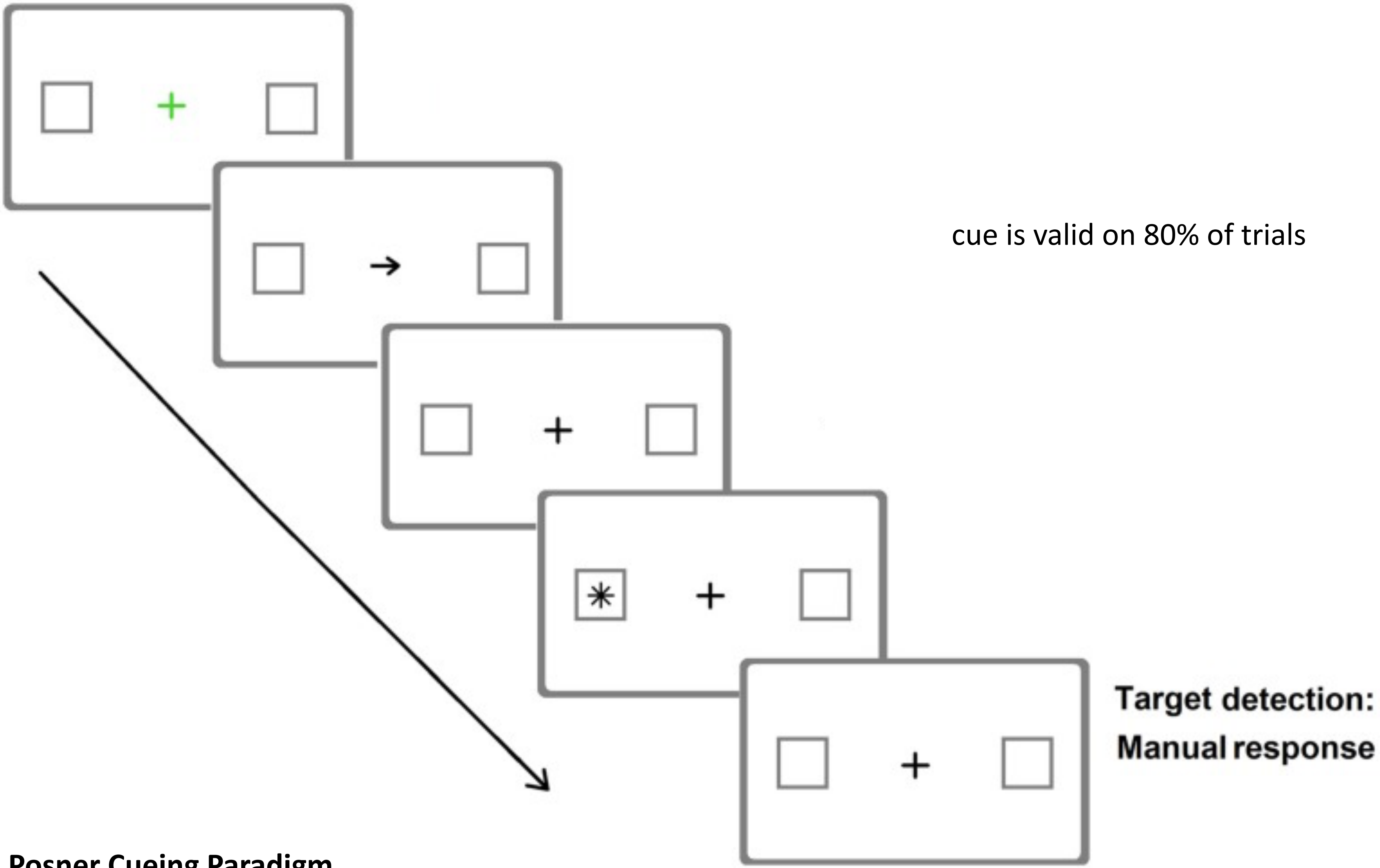
Example: simulating the gambler's fallacy

simulate gambler's fallacy (after string of 10 heads)

```
import numpy.random as R

iterations = 1000000
nheads = 0; ntries = 0; p = 0.5
run = 0
for idx in range(iterations):
    rnd = R.uniform()
    if run == 10:
        if rnd < p:
            nheads = nheads + 1
        ntries = ntries + 1
        run = 0
    else:
        if rnd < p:
            run = run + 1
        else:
            run = 0
print(nheads/ntries)
```

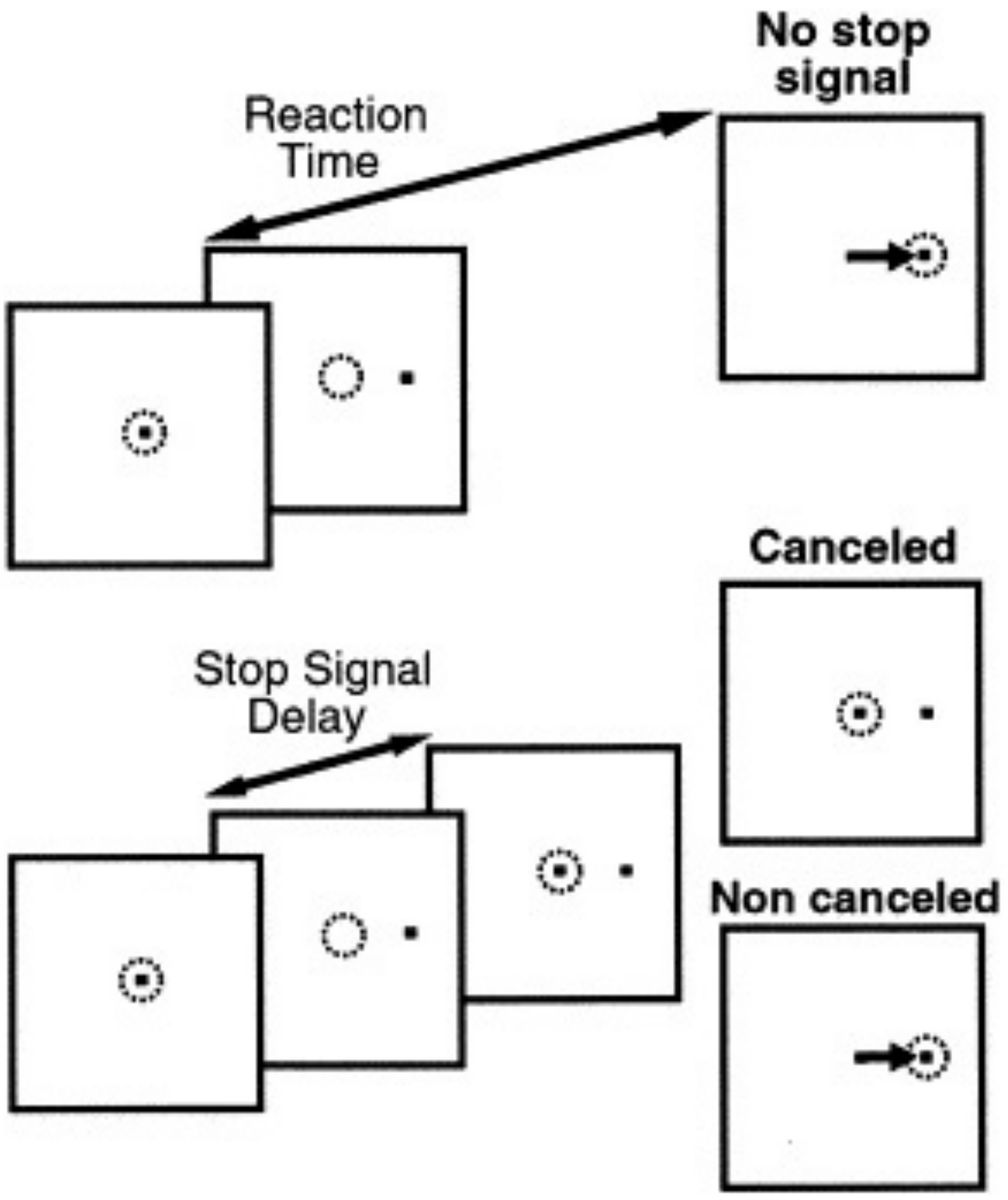
weighted "coin flips" in psychology and neuroscience



cue is valid on 80% of trials

Posner Cueing Paradigm

weighted "coin flips" in psychology and neuroscience



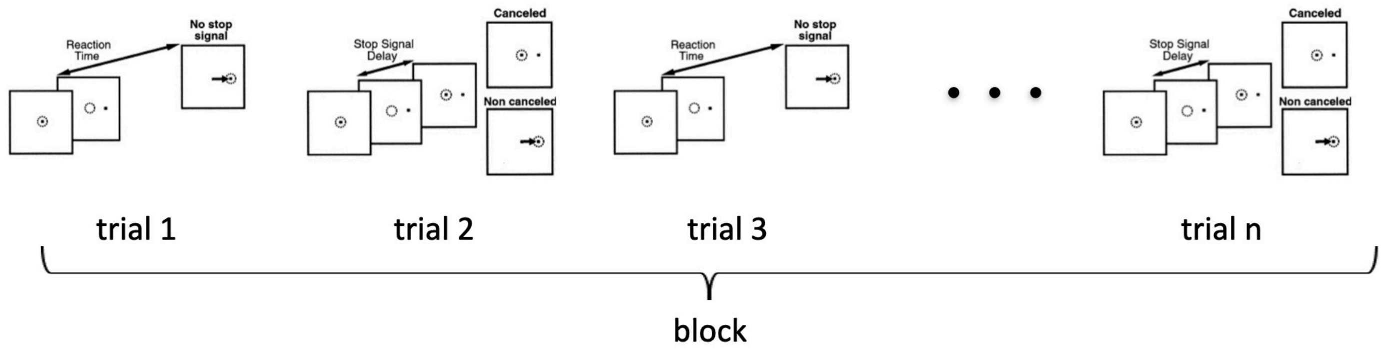
"go trial" on
70% of trials

"stop trial" on
30% of trials

Stop Signal Task (Lappin, Logan)

a quick primer on the language of experimental design

example: stop signal task



manipulations (independent variables) are introduced by the experimenter
e.g., presence/absence of stop signal, stop signal delay (ms)

levels (of an independent variable) are values of those manipulations
e.g., 50ms, 100ms, 200ms, etc. stop signal delay

conditions are combinations of particular levels of independent variables
e.g., no stop signal, stop signal @ 50ms, stop signal @ 100ms, etc.

trial is a specific condition a subject (human/animal) experiences at a time
(experiments often have multiple trials to obtain sufficient data for analyses)

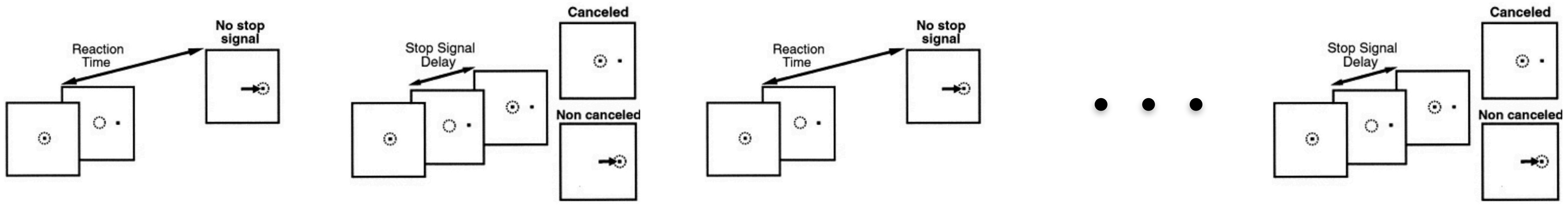
blocks are collections of trials, **sessions** are collections of blocks

within-subject means all levels are experienced by a particular subject

between-subjects means levels are experienced by different subjects

a little bit on randomizations in experiments

example: stop signal task



trial 1 trial 2 trial 3 trial n

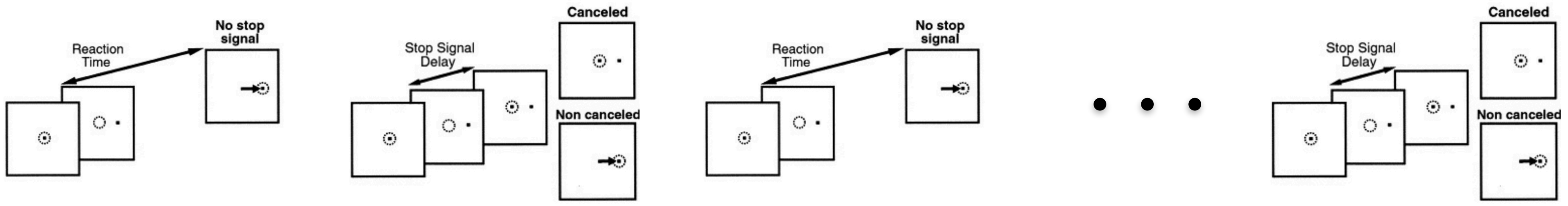
randomize to make conditions unpredictable (avoid certain strategies)

often each subject gets a different random order

(but there are times when every subject gets the same random order)

a little bit on randomizations in experiments

example: stop signal task

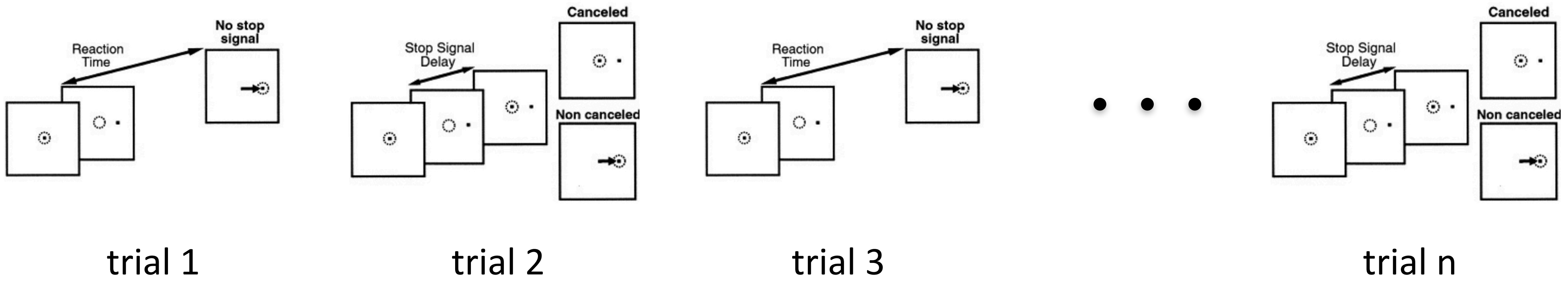


trial 1 trial 2 trial 3 trial n

how to we randomize the condition used on each trial?

a little bit on randomizations in experiments

example: stop signal task



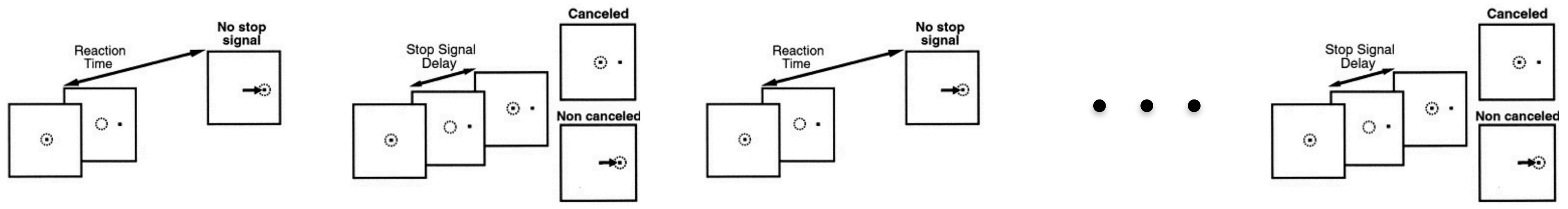
how to we randomize the condition used on each trial?

1) every trial is a (weighted) random coin flip (Bernoulli sample)

with a small number of trials, there is a good chance that the actual proportion of trials could deviate from their probability p

Homework 6 (Q3)

example: stop signal task



trial 1 trial 2 trial 3 trial n

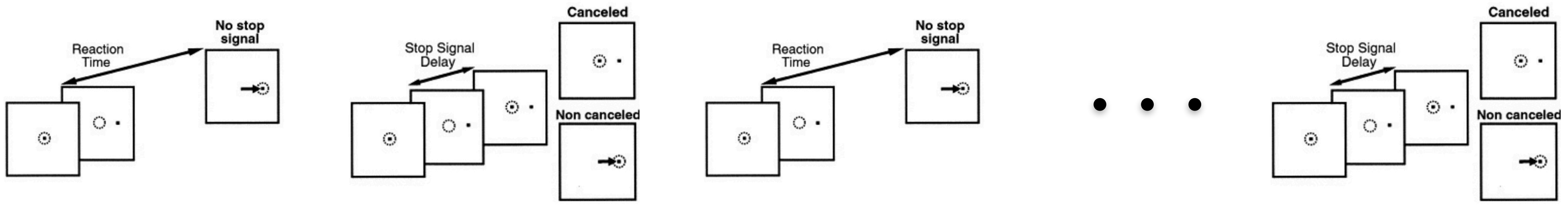
create a randomization of conditions in a stop signal task

GO with probability p (no stop signal)

STOP with probability 1-p (stop signal)

Homework 6 (Q3)

example: stop signal task



trial 1 trial 2 trial 3 trial n

create a randomization of conditions in a stop signal task

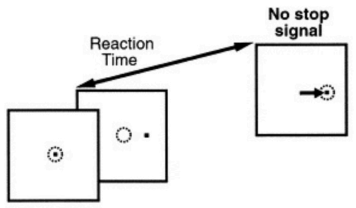
GO with probability p (no stop signal)

STOP with probability 1-p (stop signal)

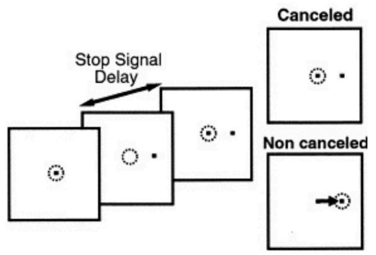
equal probability of each possible stop signal delay SSD

Homework 6 (Q3)

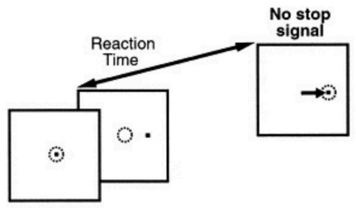
example: stop signal task



trial 1

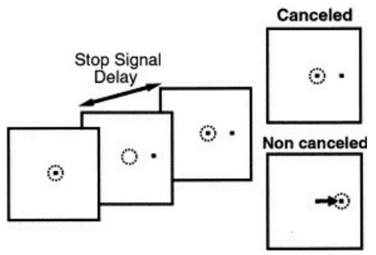


trial 2



trial 3

• • •



trial n

create a randomization of conditions in a stop signal task

GO with probability p

STOP with probability 1-p

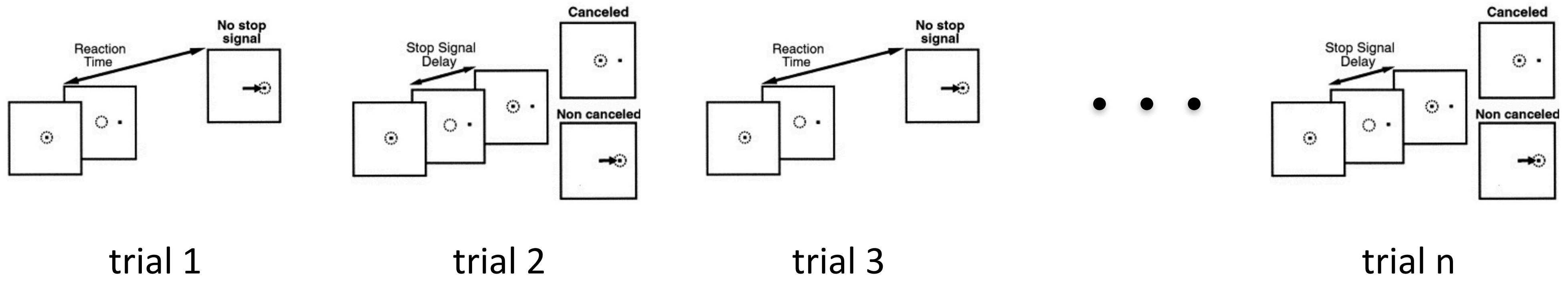
Bernoulli (weighted coin flip)

equal probability of each possible stop signal delay SSD

Bernoulli (weighted coin flip)

Homework 6 (Q3)

example: stop signal task



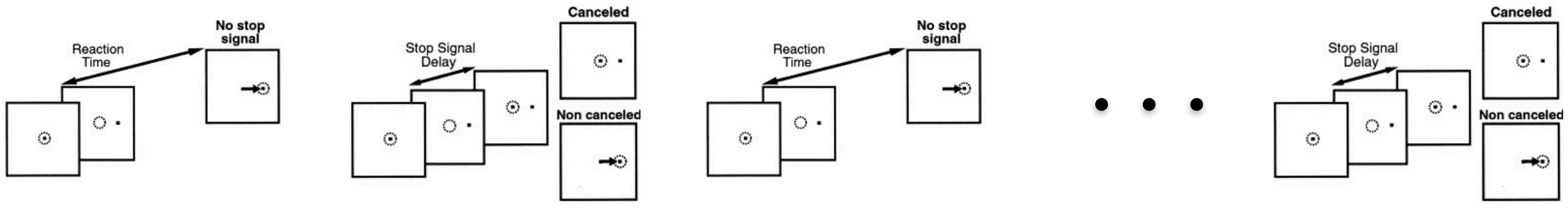
create a randomization of conditions in a stop signal task

output should be conditions (-1 = GO, otherwise=SSD)

[-1, 50, -1, -1, -1, 150, -1, 50, -1, -1, -1, -1, 50, -1, -1, 100, -1, -1, -1 ...

Homework 6 (Q3)

example: stop signal task



trial 1 trial 2 trial 3 trial n

create a randomization of conditions in a stop signal task

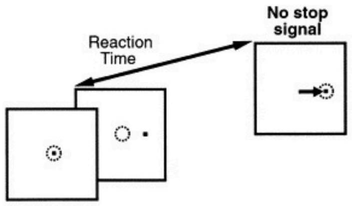
output should be conditions (-1 = GO, otherwise=SSD)

[-1, 50, -1, -1, -1, 150, -1, 50, -1, -1, -1, -1, 50, -1, -1, 100, -1, -1, -1 ...

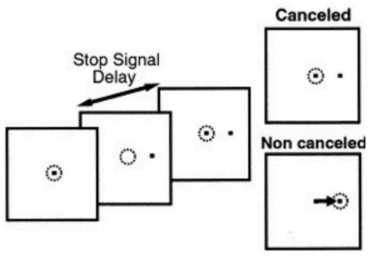
with a large number of simulated trials, <u>confirm randomization</u> works

a little bit on randomizations in experiments

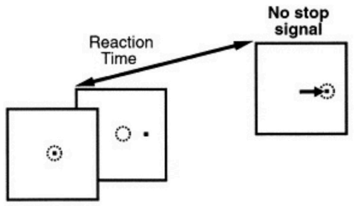
example: stop signal task



trial 1

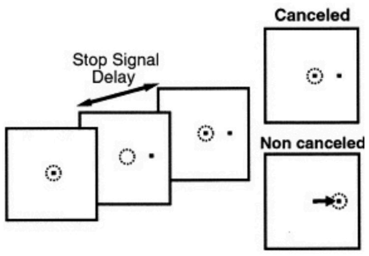


trial 2



trial 3

• • •

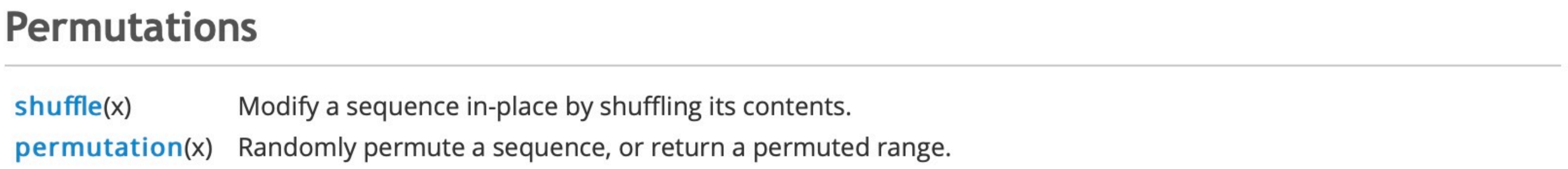


trial n

how to we randomize the condition used on each trial?

2) what if we wanted to ensure that out of every 100 trials,
70 were GO trials and 30 were STOP trials?

which can't be guaranteed with Bernoulli coin flips



Permutations

shuffle(x)	Modify a sequence in-place by shuffling its contents.
permutation(x)	Randomly permute a sequence, or return a permuted range.

imagine these are conditions in an experiment

```
a = np.array([1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3,
              3, 3, 3, 3])
```

this produces a random order (a permutation) - changes `a`

```
R.shuffle(a)
```

this produces a permutation - returns a new array

```
b = R.permutation(a)
```

Homework 6 (Q4)

implement a particular permutation algorithm

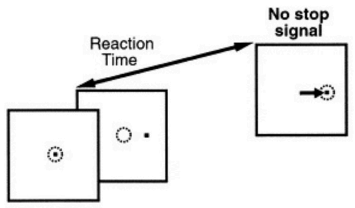
Of the various algorithms for doing permutations described on this Wikipedia page, I want you to implement the first (and oldest) one called the Fisher–Yates Shuffle. As described on the Wikipedia page, the algorithm consists of the following steps:

1. Write down the numbers from 1 through N.
2. Pick a random number k between one and the number of unstruck numbers remaining (inclusive).
3. Counting from the low end, strike out the kth number not yet struck out, and write it down at the end of a separate list.
4. Repeat from step 2 until all the numbers have been struck out.
5. The sequence of numbers written down in step 3 is now a random permutation of the original numbers.

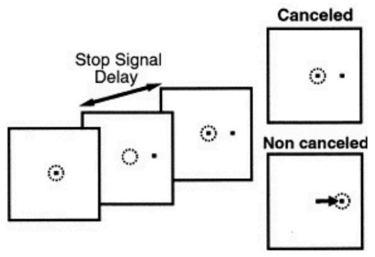
While the packages available for Python provide a lot of functions for solving a wide range of problems sometimes you need to implement an explicit algorithm for yourself.

a little bit on randomizations in experiments

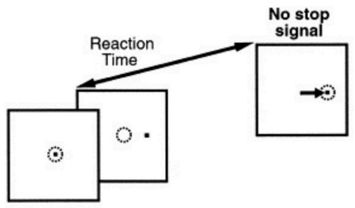
example: stop signal task



trial 1

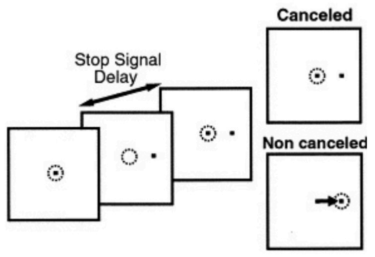


trial 2



trial 3

• • •



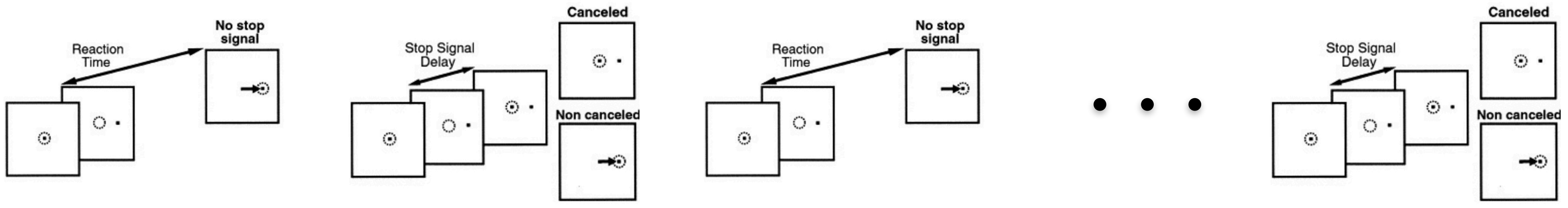
trial n

how to we randomize the condition used on each trial?

if we had 1000 trials, 700 GO and 300 STOP, there would be a
a chance that many of the STOP trials would be at the beginning

a little bit on randomizations in experiments

example: stop signal task



trial 1 trial 2 trial 3 trial n

how to we randomize the condition used on each trial?

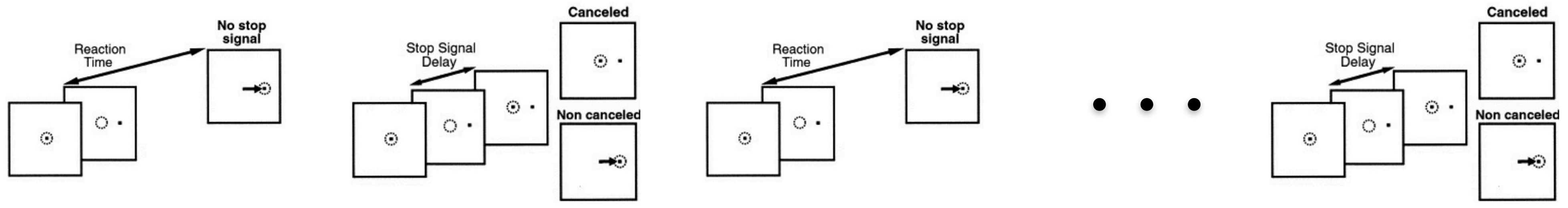
3) use a block structure (100 trials per block, permuting within a block)

might allow a rest (and give feedback) for the subject between blocks

if we had 1000 trials, 700 GO and 300 STOP, there would be a
a chance that many of the STOP trials would be at the beginning

a little bit on randomizations in experiments

example: stop signal task



trial 1 trial 2 trial 3 trial n

how to we randomize the condition used on each trial?

4) more constraints: for example, a random (weighted) coin flip, but constrained to not allow more than 8 trials of the same type in a row

example of constrained randomization

imagine having N kinds of trials (conditions), but you want to constrain so that the same condition never repeats

version 1 : just a coin flip, but no attempt to equate the number of times each condition is presented (with no repeats)

`Random.ipynb`

example of constrained randomization

imagine having N kinds of trials (conditions), but you want to constrain so that the same condition never repeats

version 1 : just a coin flip, but no attempt to equate the number of times each condition is presented (with no repeats)

version 2 : each condition needs to be presented once before a condition can be presented again (with no repeats)

`Random.ipynb`

example of constrained randomization

imagine having N kinds of trials (conditions), but you want to constrain so that the same condition never repeats

version 1 : just a coin flip, but no attempt to equate the number of times each condition is presented (with no repeats)

version 2 : each condition needs to be presented once before a condition can be presented again (with no repeats)

version 3: each condition must be presented an equal number of times within a block (with no repeats)

Homework 6 (REQ: Grads / EC: Undergrads)

different ways of generating random numbers in Python

1) using the `random` module in base Python - NOT RECOMMENDED

2) `random` module in numpy, using **seed/state-based approach**
   - what has been used for many years
   - what you find across the web
   - what most existing Python code uses
   - not "pythonic" in style
   - does not work with parallel code
   - frozen, considered legacy
   - but it is what I will start with

3) `random` module in numpy, using **generator object approach**
   - what we will talk about later
   - recommended for code going forward

`numpy`, `scipy` use generator objects, `scikit-learn` uses seed/state approach, `keras/tensorflow` use their own random number generators

generator object approach

```
import numpy.random as R

# seed a generator (for random numbers)

# PCG64() is one particular random number algorithm
# others include MT19937() for Mersenne Twister

seed = 4233981212398
rng = R.Generator(R.PCG64(seed))

print('uniform: ', rng.uniform(low=4, high=12))
```

see `Random.ipynb`